NANTUCKET CLIPPER

5

Title: The Guide To Clipper 5.0
Credits: (C)opyright 1990 Nantucket GmbH Germany
Erstellt aus der Teilen der Orginaldokumentation von F.Goeder
* = Nicht empfohlen
. = Ge.ndert gegen.ber der Original-Dokumentation
. = Neu gegen.ber der Original-Dokumentation

1. Sprache

1. Programme

1. Tabellen

RMAKE-Optionen

Die in diesem Abschnitt beschriebenen Optionen beeinflussen die Verhaltensweise von RMAKE. Alle Optionen k.nnen entweder als kleine oder als gro.e Buchstaben angegeben werden und m.ssen entweder durch einen Slash (/) oder durch einen Bindestrich eingeleitet werden. Optionen d.rfen in jeder beliebigen Reihenfolge angegeben werden. Dabei k.nnen diese Optionen in der RMAKE-Kommandozeile oder aber .ber die RMAKE-Umgebungsvariable .bergeben werden. Optionen in der Kommandozeile haben aber immer Vorrang vor denen in der Umgebungsvariablen. Einige Optionen verf.gen zus.tzlich .ber Argumente. In einem solchen Fall wird die jeweilige Syntax in der folgenden .bersicht nach der .berschrift einer speziellen Option angegeben. Zwischen Optionen und ihren Argumenten d.rfen keinerlei Leerzeichen verwendet werden.

Die Angabe von RMAKE-Optionen

RMAKE.EXE stellt die Umgebungsvariable RMAKE zur Verf.gung, damit Optionen festgelegt werden k.nnen, ohne diese jedesmal in der Kommandozeile angeben zu m.ssen. Um die RMAKE-Umgebungsvariable anzugeben, verwenden Sie das DOS SET-Kommando in folgender Weise:

```
SET RMAKE=[[<Make-Dateiliste>][<Optionen Liste>]]
```

Wird eine RMAKE-Umgebungsvariable angelegt, so werden alle Optionen genau so angegeben, wie dies innerhalb der RMAKE-Kommandozeile geschehen w.rde. Beachten Sie dabei bitte, da. alle Optionen und Dateinamen in der Umgebungsvariablen vor den Angaben in der Kommandozeile eingelesen werden. Damit haben die Angaben in der Kommandozeile Vorrang vor denen in der Umgebungsvariablen.

Um dieses DOS SET-Kommando nicht nach jedem Systemstart eingeben zu m.ssen, k.nnen Sie es in der Datei AUTOEXEC.BAT eintragen, die nach jedem Bootvorgang ausgef.hrt wird.

Beispiel
Das folgende Beispiel veranla.t RMAKE dazu, sowohl Warnungen als auch Debugging-Informationen anzuzeigen.

```
SET RMAKE=/W/B
```

Hinweis
DOS erlaubt kein Gleichheitszeichen (=) als Teil der Zuweisung an eine Umgebungs-Variable, wodurch es unm.glich wird, Makros in der gleichen Weise wie in der Kommandozeile anzulegen. Dieses Problem kann gel.st werden, indem Makros .ber die Option /D innerhalb der Umgebungsvariable definiert werden. Wie die Makros aus der Kommandozeile k.nnen auch solche aus der Umgebungsvariablen in einer Make-Datei nicht ver.ndert werden.

/B Debugging-Informationen

Diese Option veranla.t die Anzeige von Debugging-Informationen.

/D Definition eines Makros

/D<Makroname>:<Wert>

Stellt eine Alternative f.r die Definition von Makro-Variablen in der Kommandozeile durch <Makroname>=<Wert> dar. Diese Option ist f.r die Verwendung in der RMAKE-Umgebungsvariablen vorhanden, in der, bedingt durch das DOS-Kommando SET, keine Gleichheitszeichen (=) verwendet werden d.rfen.

/I Ignorieren von Ausf.hrungsfehlern

Veranla.t RMAKE fortzufahren, obwohl die Ausf.hrung einer Anweisung zu einem Fehler gef.hrt hat. Dies bedeutet, da. ein Programm einen DOS Return-Code gr..er Null geliefert hat. Als Voreinstellung stoppen Fehler dieser Art den Make-Proze..

Beachten Sie bitte, da. schwerwiegende Fehler w.hrend der Ausf.hrung von RMAKE einen Make-Proze. immer beenden, auch wenn die Option /I angegeben wurde.

/N Null Make

Zeigt alle Anweisungen nur an, anstatt sie auszuf.hren.

/S Unterverzeichnisse (Subdirectories) durchsuchen

Normalerweise durchsucht RMAKE nur das angemeldete und die explizit angegebenen Verzeichnisse nach den angegebenen Make-Dateien. Diese Option hingegen veranla.t RMAKE, auch alle Unterverzeichnisse des angemeldeten oder der explizit angegebenen Verzeichnisse nach Make-Dateien zu durchsuchen. So werden bei Aufruf von RMAKE MYMAKE /S das angemeldete und alle Unterverzeichnisse durchsucht, w.hrend bei Angabe von RMAKE
Unterverzeichnisse durchsucht werden.

/U Kommentar-Zeichen # zulassen

.ber diese Option wird die Benutzung des Kommentar-Zeichens # zugelassen. Wenn diese Option verwendet wird, m.ssen alle RMAKE-Direktiven durch ein Ausrufungszeichen anstelle des #-Zeichens eingeleitet werden.

/W Warnungen anzeigen

Zeigt w.hrend der Ausf.hrung eventuelle Warnungen an.

/XS Gr..e der Symbol-Tabelle setzen

/XS<AnzahlSymbole>

Bestimmt die Gr..e der internen Symbol-Tabelle als Anzahl von Symbolen.
Als Voreinstellung sind 500 Symbole m.glich.

/XW Gr..e des Workspace setzen

/XW<AnzahlByte>

Setzt die Gr..e des internen Arbeitsbereiches in Byte. Die vorgegebene Gr..e ist
2048 Byte.

Die RMAKE Return-Codes

RMAKE liefert einen DOS Return-Code 0, wenn die Ausf.hrung einer Make-Datei erfolgreich war, also keinerlei Fehler aufgetreten sind. Sind hingegen Fehler aufgetreten, wird der Wert 1 zur.ckgegeben. Dieser R.ckgabewert kann dann zum Beispiel innerhalb einer aufrufenden Batch-Datei .ber ERRORLEVEL gepr.ft werden. Als Beispiel dient folgen der Auszug aus einer Batch-Datei:

```
RMAKE RL
IF ERRORLEVEL 1 GOTO FEHLER
.
.<Befehle>
.
GOTO ENDE
:FEHLER
.
.<Befehle>
.
:ENDE
```

Wie RMAKE arbeitet

Falls Sie bereits andere Make-Systeme kennen, werden Sie es einfach finden, RMAKE zu verstehen und damit zu arbeiten. Grunds.tzlich ist RMAKE aufw.rts-kompatibel zu UNIX Make-Systemen und kann alle dort verwendeten Make-Dateien ausf.hren, bietet jedoch dar.ber hinaus einige zus.tzliche M.glich-keiten. Generell beschreiben Sie Ihr Programm in einer Make-Datei durch sogenannte Abh.ngigkeitsregeln. Danach benutzen Sie RMAKE in Verbindung mit einer oder mehreren f.r das jeweilige Programm-System entworfenen Make-Dateien, anstatt jedesmal alle Schritte f.r die Compilierung und das Linken nach Bedarf separat aufzurufen. RMAKE f.hrt nur die wirklich notwendigen Compilier- und Link-Schritte aus und bringt das Programm damit schnellst-m.glich auf den neuesten Stand.

Wichtigstes Element von Make-Systemen sind die Abh.ngigkeitsregeln, die ein Abh.ngigkeits-Statement enthalten, .ber welches die Beziehungen zwischen allen abh.ngigen und der Zieldatei formuliert werden. Dem Abh.ngigkeits-Statement folgt dann eine Gruppe von Anweisungen, .ber welche die Zieldatei bearbeitet wird, wenn eine der abh.ngigen Dateien neuer ist.

RMAKE ist ein Two Pass-System, es durchl.uft Make-Dateien also zweimal. Der erste Durchlauf wird Parser-Phase genannt, in ihr werden alle Definitionen .berpr.ft. Die zweite Phase, auch Make-Phase genannt, .berpr.ft alle Abh.ngigkeitsregeln und f.hrt gegebenenfalls die zugeh.rigen Anweisungen aus. Nach seinem Aufruf f.hrt RMAKE zuerst die Parser-Phase durch, wobei alle Make-Dateien in der angegebenen Reihenfolge abgearbeitet werden. Dateien aus der Kommandozeile werden von links nach rechts bearbeitet. Die Suche nach Dateien wird ebenfalls w.hrend der Parser-Phase durchgef.hrt, und die Schlu.folgerungs-Regeln werden angewendet. Werden Makros oder Regeln umdefiniert, so gelten die neuen Definitionen f.r alle darauf folgenden Abh.ngigkeiten. Nach der Parser-Phase f.hrt RMAKE die Make-Phase aus, in der die definierten Regeln aller Make-Dateien angewendet werden, um die Zieldateien zu erstellen. F.r jede Regel in einer Make-Datei wird das Datum der Zieldatei mit dem Datum jeder abh.ngigen Datei verglichen. Verf.gt eine abh.ngige Datei .ber ein neueres Datum oder eine neuere Uhrzeit als die zugeh.rige Zieldatei, so werden die unmittelbar folgenden Anweisungen ausgef.hrt. Wurde keine der abh.ngigen Dateien seit dem letzten Durchlauf ge.ndert, so wird der komplette Block .bergangen und der gesamte Vorgang f.r die n.chste Regel durchgef.hrt.

RMAKE wird auch solche F.lle korrekt behandeln, bei denen in mehreren Make-Dateien Abh.ngigkeiten (und zugeh.rige Anweisungen) f.r dieselbe Zieldatei definiert sind. Intern wird eine Liste aller Zieldateien verwaltet, um sicherzustellen, da. auch mehrfach angegebene Zieldateien (z.B. LIB-Dateien) kein neues Datum und keine neue Uhrzeit erhalten, bevor die Abh.ngigkeitsregeln aller Make-Dateien abgearbeitet wurden. Obwohl beim Einsatz von RMAKE keine bestimmte Reihenfolge der Bearbeitung garantiert werden kann, ist sehr wohl garantiert, da. keine Zieldatei neu angelegt wird, bevor alle zugeh.rigen Abh.ngigkeitsregeln bearbeitet wurden.

Sobald RMAKE beendet wird, setzt es einen DOS Return-Code, um damit den Status des durchgef.hrten Make-Prozesses anzuzeigen.

Wie RMAKE nach Dateien sucht

Es gibt nur wenige Arten von Dateien, die als Eingabe f.r RMAKE dienen. Dieser Abschnitt beschreibt jeden Typ von Eingabe-Datei und erkl.rt, nach welchen Regeln RMAKE diese Art von Datei sucht.

Make-Dateien

Sobald RMAKE aufgerufen wird, sucht es zuerst nach den Make-Dateien aus der Kommando-Zeile, dann nach denen aus der Umgebungsvariablen RMAKE. Als Voreinstellung wird vorausgesetzt, da. diese Dateien .ber die Extension .RMK verf.gen, es kann jedoch auch eine andere Datei-Extension als Teil des Make-Dateinamens angegeben werden. Um eine Datei ganz ohne Extension anzugeben, mu. ein Punkt (.) am Ende des Dateinamens angef.gt werden.

Ein vollst.ndiger oder auch teilweiser Pfad kann als Bestandteil eines Make-Dateinamens verwendet werden, um RMAKE so zu veranlassen, die Make-Datei auf dem angegebenen Laufwerk und im angegebenen Pfad zu suchen. Wurde kein Pfad spezifiziert, sucht RMAKE nur im angemeldeten Verzeichnis nach Make-Dateien. Wurde allerdings die Option /S verwendet, werden auch alle Unterverzeichnisse durchsucht.

Abh.ngige und Ziel-Dateien

Neben den Make-Dateien mu. RMAKE auch die abh.ngigen, sowie die Ziel-Dateien aus den Abh.ngigkeitsregeln finden k.nnen. Als Voreinstellung werden diese Dateien ausschlie.lich im angemeldeten Verzeichnis gesucht. Dar.ber hinaus existieren jedoch spezielle Makros, .ber die festgelegt werden kann, wo die abh.ngigen und die Ziel-Dateien gefunden werden k.nnen, bzw. angelegt werden sollen. Die Form dieser speziellen Makro-Definition ist:

makepath[.<Extension>] = <Pfad>

<Extension> gibt den Typ der Datei und <Pfad> einen oder mehrere Pfade an, in denen zu suchen ist, wenn eine solche Datei gefunden oder angelegt werden soll. Um Probleme mit dem Backslash (\) zu vermeiden, sollten Pfade, wie im folgenden Beispiel, immer in doppelten Anf.hrungszeichen angegeben werden.

Wenn ein makepath-Makro definiert wurde, wird das angemeldete Verzeichnis nicht mit durchsucht, wenn es nicht explizit als Teil der Pfad-Liste angegeben wurde. Nur die angegebenen Pfade werden w.hrend der Parser-Phase in der Reihenfolge durchsucht, in der sie spezifiziert wurden.

Wenn die gesuchte Datei in keinem der angegebenen Pfade gefunden werden konnte, so kombiniert RMAKE den ersten Pfad aus der Pfadliste <Pfad> mit den Namen der Zieldateien. So werden die Zieldateien in genau diesem Pfad angelegt. Kann RMAKE hingegen abh.ngige Dateien nicht finden, so f.hrt dies zu einem Fehler.

Die makepath-Makros k.nnen in der RMAKE-Kommandozeile, .ber die RMAKE-Umgebungsvariable (unter Benutzung der Option /D) oder aber in einer Make-Datei definiert werden. Hier einige Beispiele dazu:

```
makepath[.h]   = "C:\CLIPPER5\SOURCE;C:\CLIPPER5\INCLUDE"
makepath[.obj] = "C:\CLIPPER5\OBJ"
makepath[.lib] = "C:\CLIPPER5\LIB;C:\MSC\LIB"
makepath[.exe] = "C:\CLIPPER5\BIN"
```

Um eine Pfadliste zu vergr..ern, benutzen Sie bitte den immediate assignment Operator (:=) anstelle des assignment Operators (=):

```
makepath[.h] := "C:\MSC\INCLUDE;$(makepath[.h])"
```

Hinweis

Alle Anweisungen werden erst w.hrend der Make-Phase ausgef.hrt und haben somit keinen Einflu. auf das Suchen von Dateien. So k.nnen Sie zum Beispiel

keine Anweisung verwenden, die eine Datei in einem bestimmten, durch einen Pfad angegebenen Verzeichnis ablegt, um so eine bestimmte Abh.ngigkeits-regel zu beeinflussen. Bei allen abh.ngigen und Zieldateien, die zusammen mit einem Pfad angegeben werden, verwendet RMAKE nur diesen Pfad zur Suche und l..t alle Angaben f.r das makepath-Makro unber.cksichtigt.

Ein ausf.hrlicheres Beispiel

Das folgende Beispiel zeigt eine Make-Datei, .ber die RL.EXE, das Clipper Report- und Label-System, compiliert und gelinkt werden kann. Dieses Programm besteht aus den folgenden Quell-Code Modulen:

. Rlfront.prg
. Rlback.prg
. Rldialog.prg

Um RL.EXE zu erzeugen, wird jede der Quell-Dateien in eine Objekt-Datei compiliert und diese dann mit Hilfe des Linkers zu der .EXE-Datei zusammengebunden. Die folgende Make-Datei legt alle notwendigen Regeln sowie die zugeh.rigen Anweisungen fest:

```
// RL.RMK
// Makros definieren
objDat=RLFRONT, RLBACK, RLDIALOG
//
// Schlu.folgerungsregel: Compilierung .PRG- zu .OBJ-Dateien
.prg.obj:
  CLIPPER $** /M
//
// Abh.ngigkeitsregeln f.r .OBJ-Dateien
RLFRONT.OBJ: Rlfront.prg
RLBACK.OBJ: Rlback.prg
RLDIALOG.OBJ: Rldialog.prg
//
// Abh.ngigkeitsregel: .OBJ-Dateien zu .EXE-Datei linken
RL.EXE: RLFRONT.OBJ RLBACK.OBJ RLDIALOG.OBJ
  RTLINK OUTPUT $@ FI $(objDat)
```

Vorausgesetzt, diese Make-Datei wurde RL.RMK genannt, kann RL.EXE nach .nderungen in einer der Quell-Dateien durch das folgende DOS-Kommando neu erstellt werden:

```
RMAKE RL [Return]
```

Hinweise

. L.nge der DOS-Kommandozeile: Da RMAKE die DOS-Kommandos in verschiedenen F.llen auch durch eine Expansion von Makros erzeugt, besteht die Gefahr, da. solche Zeilen l.nger als das unter DOS zul.ssige Limit von 128 Zeichen werden k.nnten. Dies kann beispielsweise zu einem Problem werden, wenn zu einer ausf.hrbaren Datei eine gro.e Anzahl von Objekt-Dateien geh.ren und die Anweisungszeile dadurch letztendlich zu lang wird. In einem solchen Fall sollte eine Script-Datei f.r RTLink (.lnk) verwendet werden, innerhalb derer alle abh.ngigen Objekt-Dateien angegeben sind, und die in der Anweisungszeile anstelle der Liste von Objekt-Dateien eingesetzt wird. Genauere Informationen .ber die Benutzung von Script-Dateien finden Sie im Kapitel Clipper-Linker - RTLINK.EXE innerhalb dieses Handbuches.

. Eine Schlu.folgerungsregel in Verbindung mit einer Kommandozeile des Compilers benutzen: Wird eine Schlu.folgerungsregel eingesetzt, um den Clipper-Compiler aufzurufen, so ist darauf zu achten, da. nicht mehr als eine abh.ngige Datei in der zugeh.rigen Abh.ngigkeitsregel angegeben wird. Anderenfalls w.rde zumindest bei der Verwendung des vordefinierten Makros $** die Liste aller abh.ngigen Dateien eingesetzt, und der Compiler w.rde einen Fehler melden, da er immer nur eine Quell-Datei als Argument entgegennehmen kann.

Um mit dem Clipper-Compiler mehrere Quell-Dateien zu einer Objekt-Datei zu compilieren, wird am besten eine Compiler Script-Datei eingesetzt, welche alle Quell-Dateien enth.lt, aus denen die Objekt-Datei erstellt

werden soll. Um diese Technik m.glichst einfach nutzen zu k.nnen, geben Sie der Script-Datei m.glichst denselben Namen wie der Objekt-Datei, die erzeugt werden soll, allerdings mit der .CLP-Extension. Wenn Sie dann die Make-Datei daf.r erzeugen, legen Sie die Abh.ngigkeits- und Schlu.folgerungsregeln wie folgt fest:

```
// Standard Schlu.folgerungsregel
// f.r das Compilieren .ber eine Script-Datei
.prg.obj:
  CLIPPER @$*
//
// Typische Abh.ngigkeitsregel zur Erzeugung einer Objekt-Datei
TEST.OBJ: Test1.prg Test2.prg Test3.prg
```

Die Datei Test.clp m..te in diesem Fall die drei .PRG-Dateien enthalten, aus denen die Objekt-Datei erstellt werden soll. Die Kommando-Zeile des Compilers wird dann unter Benutzung des vordefinierten Makros $* erzeugt, welches nur den Namen der Zieldatei erg.nzt. Daraus resultiert letztendlich das Kommando CLIPPER @TEST, sobald die Schlu.folgerungsregel ausgef.hrt wird. Das Zeichen @ weist Clipper an, die Namen der zu compilierenden Quell-Dateien aus der Datei Test.clp einzulesen.
Weitere Informationen .ber Script-Dateien des Clipper-Compilers finden Sie im Kapitel Der Clipper-Compiler innerhalb dieses Handbuches.
Zus.tzliche Informationen .ber vordefinierte Makros sind im vorherigen Abschnitt zu finden.

Cursor-Bewegungstasten

				Editiertasten		
PfeilAufw.rts	Strg+E	5	K_UP	Entf	7	K_DEL
PfeilAbw.rts	Strg+X	24	K_DOWN	Tab	9	K_TAB
Linkspfeil	Strg+S	19	K_LEFT	Umschalt+Tab	271	K_SH_TAB
Rechtspfeil	Strg+D	4	K_RIGHT	Einfg, Strg+V	22	K_INS
Pos1	Strg+A	1	K_HOME	Strg+R.ckschritt	127	K_STRG_BS
Ende	Strg+F	6	K_END	R.ckschritt	8	K_BS
Bild.	Strg+R	18	K_PGUP			
Bild.	Strg+C	3	K_PGND	**Funktionstasten**		
Strg+Linkspfeil	Strg+Z	26	K_CTRL_LEFT	F1, Strg+\	28	K_F1
Strg+Rechtspfeil	Strg+B	2	K_CTRL_RIGHT	F2	-1	K_F2
Strg+Pos1	Strg+]	29	K_CTRL_HOME	F3	-2	K_F3
Strg+Ende	Strg+W	23	K_CTRL_END	F4	-3	K_F4
Strg+Bild.	Strg+-	31	K_CTRL_PGUP	F5	-4	K_F5
Strg+Bild.	Strg+^	30	K_CTRL_PGDN	F6	-5	K_F6
Strg+Return		10	K_CTRL_RET	F7	-6	K_F7
Esc	Strg+[	27	K_ESC	F8	-7	K_F8
Return	Strg+M	13	K_RETURN	F9	-8	K_F9
Eingabe	Strg+M	13	K_ENTER	F10	-9	K_F10

Alt- und Kontrolltasten

ALT+A	286	K_ALT_A	ALT+N	305	K_ALT_N	Strg+A	1	K_CTRL_A	Strg+N	14	K_CTRL_N
ALT+B	304	K_ALT_B	ALT+O	280	K_ALT_O	Strg+B	2	K_CTRL_B	Strg+O	15	K_CTRL_O
ALT+C	302	K_ALT_C	ALT+P	281	K_ALT_P	Strg+C	3	K_CTRL_C	Strg+P	16	K_CTRL_P
ALT+D	288	K_ALT_D	ALT+Q	272	K_ALT_Q	Strg+D	4	K_CTRL_D	Strg+Q	17	K_CTRL_Q
ALT+E	274	K_ALT_E	ALT+R	275	K_ALT_R	Strg+E	5	K_CTRL_E	Strg+R	18	K_CTRL_R
ALT+F	289	K_ALT_F	ALT+S	287	K_ALT_S	Strg+F	6	K_CTRL_F	Strg+S	19	K_CTRL_S
ALT+G	290	K_ALT_G	ALT+T	276	K_ALT_T	Strg+G	7	K_CTRL_G	Strg+T	20	K_CTRL_T
ALT+H	291	K_ALT_H	ALT+U	278	K_ALT_U	Strg+H	8	K_CTRL_H	Strg+U	21	K_CTRL_U
ALT+I	279	K_ALT_I	ALT+V	303	K_ALT_V	Strg+I	9	K_CTRL_I	Strg+V	22	K_CTRL_V
ALT+J	292	K_ALT_J	ALT+W	273	K_ALT_W	Strg+J	10	K_CTRL_J	Strg+W	23	K_CTRL_W
ALT+K	293	K_ALT_K	ALT+X	301	K_ALT_X	Strg+K	11	K_CTRL_K	Strg+X	24	K_CTRL_X
ALT+L	294	K_ALT_L	ALT+Y	277	K_ALT_Y	Strg+L	12	K_CTRL_L	Strg+Y	25	K_CTRL_Y
ALT+M	306	K_ALT_M	ALT+Z	300	K_ALT_Z	Strg+M	13	K_CTRL_M	Strg+Z	26	K_CTRL_Z

Umschalt-, Kontroll- und Alt-Funktionstasten

Umschalt+F1	-10	K_SH_F1	Strg+F1	-20	K_CTRL_F1	Alt+F1	-30	K_ALT_F1
Umschalt+F2	-11	K_SH_F2	Strg+F2	-21	K_CTRL_F2	Alt+F2	-31	K_ALT_F2
Umschalt+F3	-12	K_SH_F3	Strg+F3	-22	K_CTRL_F3	Alt+F3	-32	K_ALT_F3
Umschalt+F4	-13	K_SH_F4	Strg+F4	-23	K_CTRL_F4	Alt+F4	-33	K_ALT_F4
Umschalt+F5	-14	K_SH_F5	Strg+F5	-24	K_CTRL_F5	Alt+F5	-34	K_ALT_F5
Umschalt+F6	-15	K_SH_F6	Strg+F6	-25	K_CTRL_F6	Alt+F6	-35	K_ALT_F6
Umschalt+F7	-16	K_SH_F7	Strg+F7	-26	K_CTRL_F7	Alt+F7	-36	K_ALT_F7
Umschalt+F8	-17	K_SH_F8	Strg+F8	-27	K_CTRL_F8	Alt+F8	-37	K_ALT_F8
Umschalt+F9	-18	K_SH_F9	Strg+F9	-28	K_CTRL_F9	Alt+F9	-38	K_ALT_F9
Umschalt+F10	-19	K_SH_F10	Strg+F10	-29	K_CTRL_F10	Alt+F10	-39	K_ALT_F10

See Also:

BOF() Stellt fest, ob der Dateianfang erreicht ist

Syntax

BOF() -> lDateianfang

R.ckgabe

BOF() gibt true (.T.) zur.ck, wenn Sie versuchen, den Satzzeiger vor den ersten Datensatz zu setzen. Der Satzzeiger bleibt dabei auf dem ersten Datensatz. Enth.lt die aktuelle Datenbank noch keine Datens.tze, so liefern sowohl BOF() als auch EOF() den logischen Wert true (.T.).

Anwendung

BOF() ist eine Datenbankfunktion und wird normalerweise als Test f.r solche Programme angewendet, in denen ein Satzzeiger mit dem SKIP-Befehl r.ckw.rts durch die Datenbank bewegt werden mu.. Eine einfache Anwendung ist eine Datenbank in absteigender Ordnung ohne Indexdatei. Eine etwas anspruchsvollere Anwendung ist ein Programm, das am Bildschirm Datens.tze anzeigt und in Abh.ngigkeit von einer gedr.ckten Taste den Satzzeiger durch die Datenbank bewegt. In diesem Fall kann BOF() dazu benutzt werden, um festzustellen, wann der Dateianfang erreicht haben. Ist BOF() auf true (.T.) esetzt, bleibt der Wert so lange erhalten, bis ein anderer Zugriff den Datensatzzeiger bewegt.

Normalerweise wirkt BOF() auf den momentan selektierten Arbeitsbereich. Diese Funktion kann auch auf einen nicht selektierten Arbeitsbereich angewandt werden, indem ein Aliasbezeicher vorangestellt wird (siehe Beispiel weiter unten). Der SKIP-Befehl ist die einzige Anweisung, die BOF() auf true (.T.) setzen kann.

Beispiele

. Das folgende Beispiel demonstriert das Verhalten von BOF(), wenn versucht wird, den Datensatzzeiger vor den Anfang der Datei zu stellen:

```
USE verkauf NEW
? RECNO(), BOF()  // Resultat: 1, .F.
SKIP -1
? RECNO(), BOF()  // Resultat: 1, .T.
```

. Auf BOF() in einem nicht selektierten Arbeitsbereich pr.fen:

```
USE Verkauf  NEW
USE Rechnung NEW

? Verkauf->(BOF()), Kunden->(BOF())
```

Dateien

Bibliothek ist CLIPPER.LIB

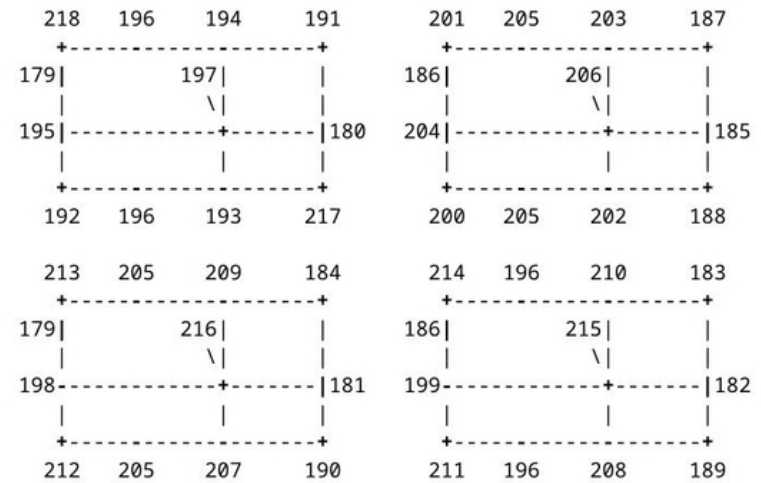
See Also:

ASCII-Zeichensatztabelle

	00	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
		.		0	@	P	`	p	.	.	.	#	+	-	.	.
00	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
	.	.	!	1	A	Q	a	q	.	.	.	#	-	-	.	.
01	1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241
	.	.	"	2	B	R	b	r	.	.	.	#	-	-	.	.
02	2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
	.	.	#	3	C	S	c	s	.	.	.	|	|	+	.	.
03	3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
	.	.	$	4	D	T	d	t	.	.	.	|	-	+	.	.
04	4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
	.	.	%	5	E	U	e	u	.	.	.	|	+	+	.	.
05	5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
	.	.	&	6	F	V	f	v	.	.	.	|	-	+	.	.
06	6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
	.	.	'	7	G	W	g	w	.	.	.	+	-	+	.	.
07	7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
	.	.	(	8	H	X	h	x	.	.	.	+	+	+	.	.
08	8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
	.	.	)	9	I	Y	i	y	.	.	.	|	+	+	.	.
09	9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
	.	.	*	:	J	Z	j	z	.	.	.	|	-	+	.	.
0A	10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
	.	.	+	;	K	[	k	{	.	.	.	+	-	#	.	.
0B	11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
	.	.	,	<	L	\	l	|	.	.	.	+	|	#	.	.
0C	12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
	.	.	-	=	M	]	m	}	.	.	.	+	-	#	.	.
0D	13	29	45	61	77	93	109	125	141	157	173	189	205	221	237	253
	.	.	.	>	N	^	n	~	.	.	.	+	+	#	.	.
0E	14	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254
	.	.	/	?	O	_	o	.	.	.	.	+	-	#	.	
0F	15	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255

Standard-Definitionen f.r Box-Kommandos, definiert in BOX.CH

```
B_SINGLE        "+-+|+-+|" +---+    B_SINGLE_DOUBLE "+-+|+-+|" +---+
                           +---+                               +---+
B_DOUBLE        "+-+|+-+|" +---+    B_DOUBLE_SINGLE "+-+|+-+|" +---+
```



Mathematische Symbole

.	238	ist ein Element von	.	246	Division
.	239	Durchschnitt	.	247	Nahezu gleich, zirka
.	240	Identisch, kongrurent	.	248	Grad
.	241	Plus oder Minus	.	249	Multiplikation
.	242	Gr..er oder gleich	.	250	
.	243	Kleiner oder gleich	.	251	Wurzel
.	244	Integral	.	252	Potenz
.	245		.	253	Zum Quadrat
.	244	Integral	.	236	Unendlich
|	179				
.	245				

Griechische Zeichen

```
.     224       Alpha
.     225       Beta
.     226       Gamma
.     235       Delta
.     233       Theta
.     230       Mu
.     227       Pi
.  .  228 229   Sigma
.     231       Tau
.  .  237 232   Phi
.     234       Omega
```

L.nderspezifische Zeichen

```
.    131          .    136          .    147          .    150
. .  132 142      .    137          .    148          . .  129 154
.    133          .    138          .    149          .    151
.    160          . .  130 144      .    162          .    163
. .  134 143      .    140          . .  164 165
. .  145 146      .    139                            .    152
. .  135 128      .    141          .    225
                  .    161
```

Farbtabelle

Farbe	Zeichen	Ziffer	Monochrom	
Schwarz	N, Leerzeichen	0	Schwarz	Test
Blau	B	1	Unterstrichen	Test
Gr.n	G	2	Wei.	Test
Cyan	BG	3	Wei.	Test
Rot	R	4	Wei.	Test
Magenta	RB	5	Wei.	Test
Braun	RG	6	Wei.	Test
Wei.	W	7	Wei.	Test
Grau	N+	8	Schwarz	Test
Hellblau	B+	9	Hell Unterstrichen	Test
Hellgr.n	G+	10	Hellwei.	Test
Hellcyan	BG+	11	Hellwei.	Test
Hellrot	R+	12	Hellwei.	Test
Hellmagenta	RB+	13	Hellwei.	Test
Gelb	RG+	14	Hellwei.	Test
Hellwei.	W+	15	Hellwei.	Test
Schwarz	U		Unterstrichen	
Inverse	I		Inverse	
Leer	X		Leer	

Clipper 5.0
(C)opyright 1990 Nantucket Corp.

Produktentwicklung

Software Design
Richard McConnell

Software Entwicklung
Dennis L.Dias
Richard McConnell

Unter wesentlicher Mitwirkung von
Brian Russell (Design und Implementierung)
Eric Jakstadt (Linker)
Cory Schwarz (Debugger)
Leonard Zerman (Installationssystem)

Software-Test
Phil Usher
Ed Bell
Woon L.Leong
Tim Wong
Fred Ho
Jon Rognerud

Beispiel-Programme
Christopher White, Debby Moody
David R.Alison, Craig Ogg
Jake Yacoub, Tim Wong, Flemming Ho

Orginal-Produktdokumentation
Christopher White
Deborah Ridolfo
Debby Moody
Savannah Brentall

Clipper 5.0 - Deutsche Version

Deutsche Produktdokumentation
Axel Fenger
Friedhelm Goeder
Astrid Kleine
Dr. Burak Kozan
Stefanie Krudewig
Frank Mantek
Miachael Peters
Frank Rose

Deutsche Software-Anpassung
Lothar Bongartz
Friedhelm Goeder
Dr. Burak Kozan
Frank Mantek
Michael Peters

Wir danken all denjenigen, die durch ihren Einsatz zur Realisierung von Clipper 5.0 beigetragen haben.

Klassen-Funktionen

Export-Instanzvariablen

Fehler-Klasse
  Bereitstellung von Objekten mit Informationen .ber Laufzeitfehler
---
Beschreibung
  Ein Fehler-Objekt ist ein einfaches Objekt, das Informationen .ber einen
  eventuell aufgetretenen Laufzeitfehler enth.lt. Fehler-Objekte besitzen
  keine Methoden, lediglich Export-Instanzvariablen. Tritt ein Fehler
  w.hrend der Laufzeit einer Applikation (runtime error) auf, wird ein
  neues Objekt erzeugt und als Parameter einer Fehlerbehandlungs-Routine
  .bergeben. Diese Fehlerbehandlungs-Routine kann .ber die Funktion
  ERRORBLOCK() definiert werden und das Fehler-Objekt befragen, was den
  Fehler verursacht hat.

  Ein Fehler-Objekt kann auch an eine RECOVER-Anweisung eines BEGIN
  SEQUENCE-Konstruktes mit einer BREAK-Anweisung .bergeben werden. In
  diesem Fall k.nnen Fehler-Objekte zur Behandlung von lokal aufgetretenen
  Fehlern befragt werden.

Beispiele

  . Dieses Beispiel zeigt, wie Fehler, die beim .ffnen der Datei entstehen
    k.nnen, behandelt werden. Wenn zum Beispiel mit dem Befehl USE
    vergeblich versucht wird, eine Datei zu .ffnen, kann mit folgenden
    Befehlszeilen reagiert werden:

```
#include "ERROR.CH"
#command RETRY      = RETURN(.T.)     // Weiter versuchen
#command RESUME     = RETURN(.F.)     // Standard-Fehlerbehandlung
//
FUNCTION MyError( objError )
  //
  // Fehlerbehandlung f.r Datei-.ffnungsfehler
  IF objError:genCode == EG_OPEN .AND.;
     objError:canDefault .AND.;
     NETERR()
     //
     RESUME
  ENDIF
  .
  . Andere Fehlerbehandlungen
  .
  RETURN NIL
```

  . Dieses Beispiel zeigt, wie eine Fehlerbehandlungsroutine eine
    bestimmte Anzahl von Wiederholungsversuchen durchf.hren kann:

```
#include "ERROR.CH"
#command RETRY      = RETURN(.T.)     // Weiter versuchen
#command RESUME     = RETURN(.F.)     // Standard-Fehlerbehandlung
//
FUNCTION MyError( objError )
  //
  // Fehlerbehandlung f.r 'Printer not ready'
  IF objError:genCode == EG_PRINT .AND.;
     objError:canRetry .AND.;
     objError:tries  25
     //
     RETRY
  ENDIF
  .
  . Andere Fehlerbehandlungen
  .
  RETURN NIL
```

  . Der folgende Programmteil demonstriert, wie ein Fehler-Objekt aus
    einer Fehlerbehandlungs-Routine heraus f.r die weitere Bearbeitung an

das RECOVER-Statement .bergeben werden kann:

```
LOCAL objLocal, bLastHandler
//
// Aktuelle Fehlerbehandlung retten und neue installieren
bLastHandler := ERRORBLOCK({ |objErr| MyHandler(objErr, .T.) })
//
BEGIN SEQUENCE
  .
  . Operation, die fehlschlagen k.nnte
  .
RECOVER USING objLocal
  .
  IF objLocal:filename == "TEST.PRG"
     . <Individuelle Fehlerbehandlung>
  ENDIF
  .
END
//
// Vorherige Fehlerbehandlung wiederherstellen
  ERRORBLOCK(bLastHandler)
RETURN

FUNCTION MyHandler( objError, lLocalHandler )
  //
  // Lokale Behandlung durch R.ckgabe der Fehler-Objektes
  IF lLocalHandler
    BREAK objError
  ENDIF
  .
  . <Andere Arten der Fehlerbehandlung>
  .
RETURN NIL
```

Dateien

Header-Datei ist Error.ch, Standard Fehlerbehandlung ist in ERRORSYS.PRG definiert

---

See Also:

ErrorNew()	Erzeugt ein neues Fehler-Objekt

. **args**

  Enth.lt ein Array mit Argumenten, die eine Operation oder Funktion mit
  den ben.tigten Werten versorgen, wenn ein Parameterfehler aufgetreten
  ist. F.r andere Arten von Fehlern enth.lt diese Instanzvariable den Wert
  NIL.

. **canDefault**

Enth.lt einen logischen Wert, der anzeigt, ob das System eine vorgegebene Wiederherstellungsroutine f.r eine Fehlerbedingung ausf.hren kann. Ein Wert true (.T.) zeigt an, da. ein vorgegebener Wiederherstellungsmechanismus verf.gbar ist. Die Verf.gbarkeit einer vorgegebenen Routine und einer aktuell vorgegebenen Wiederherstellungsstrategie ist abh.ngig vom Subsystem und von der Fehlerbedingung. Die einfachste Aktion ist, den Fehler zu ignorieren.

Die vorgegebene Wiederherstellung wird angefordert, indem von dem Code-Block, der zur Behandlung des Fehlers aufgerufen wurde, false (.F.) zur.ckgegeben wird. Beachten Sie, da. Error:canDefault nie den Wert true (.T.) annimmt, wenn Error:canSubstitute den Wert true (.T.) enth.lt.

. **canRetry**

Enth.lt einen logischen Wert, der besagt, ob das System eine Wiederholung der Operation, die den Fehler verursacht hat, durchf.hren kann. Der Wert true (.T.) zeigt an, da. eine Wiederholung m.glich ist. Ob ein Wiederholungsversuch erfolgen darf, ist abh.ngig vom Subsystem und der Fehlerbedingung.

Die Wiederholungsversuche werden angefordert, indem von dem Code-Block, der zur Behandlung des Fehlers aufgerufen wurde, true (.T.) zur.ckgegeben wird. Beachten Sie, da. Error:canRetry nie den Wert true (.T.) annimmt, wenn Error:canSubstitute den Wert true (.T.) enth.lt.

. **canSubstitute**

Enth.lt einen logischen Wert, der besagt, ob das Ergebnis einer Operation, die den Fehler verursacht hat, durch einen neuen Wert ersetzt werden kann. Parameterfehler und bestimmte andere einfache Fehler erlauben der Fehlerbehandlungs-Routine, den Wert einer fehlgeschlagenen Operation zu ersetzen. Der Wert true (.T.) zeigt an, da. eine Werteersetzung erfolgen kann.

Eine Neuzuweisung erfolgt durch R.ckgabe des neuen Wertes durch den Code-Block, der als Fehlerbehandlungs-Routine aufgerufen wurde. Beachten Sie, da. Error:canSubstitute nie den Wert true (.T.) annehmen kann, wenn Error:canDefault oder Error:canRetry auf true (.T.) gesetzt ist.

. cargo (zuweisungsf.hig)

Die Cargo-Information kann einen Wert beliebigen Typs aufnehmen. Dieser Wert wird von dem Objekt selber nicht benutzt, er steht vielmehr dem Programmierer als eine Art Huckepack-Information zur Verf.gung, auf die jederzeit aus einem Programm heraus zugegriffen werden kann.

. **description**

Enth.lt eine Zeichenkette, die die Fehlerbedingung beschreibt. Eine leere Zeichenkette ("") zeigt an, da. das Subsystem keine Fehlerbeschreibung zur Verf.gung stellen kann. Wenn Error:genCode ungleich Null ist, steht immer eine Fehlerbeschreibung zur Verf.gung.

. filename

Enth.lt eine Zeichenkette, welche den Namen der Datei enh.lt, die beim .ffnen einen Fehler verursachte. Eine leere Zeichenkette zeigt an, da. eine Fehlerbedingung nicht auf einen Dateifehler hinweist, oder da. das Subsystem keine Dateinamensinformation enth.lt.

. **genCode**

Enth.lt einen ganzzahligen, numerischen Wert, der einen Clipper-Fehlercode darstellt. Dieser Fehlercode erlaubt die Behandlung von .hnlichen, durch verschiedene Subsysteme ausgel.ste Fehler. Der Wert Null zeigt an, da. es sich um eine spezifische Fehlerbedingung eines Subsystems, und nicht um einen Clipper-Fehlercode handelt (s. subCode).

In der Header-Datei Error.ch sind eine Reihe von Konstanten definiert, die den Umgang mit Fehlercodes erleichtern.

. operation

Enth.lt eine Zeichenkette, welche die Operation beschreibt, durch die ein Fehler verursacht wurde. F.r Operanden und Funktionen enth.lt diese Variable den Namen des Operanden oder der Funktion. Bei undefinierten Variablen oder Funktionen enth.lt sie den Namen der Variablen oder der Funktion. Eine leere Zeichenkette besagt, da. das Subsystem keine darstellbare Beschreibung der Operation enth.lt.

. **osCode**

Enth.lt als ganzzahligen, numerischen Wert den Fehlercode des Betrieb-
systems, der den Fehler genauer spezifiziert. Eine Null bedeutet, da. der
Fehler nicht durch einen Betriebssystemfehler verursacht worden ist.

Wenn Error:osCode einen Wert gr..er Null enth.lt, wird DOSERROR() mit
dem gleichem Wert aktualisiert.

. **subCode**
Enth.lt einen ganzzahligen numerischen Wert eines vom Subsystem spezifizierten Fehler-Codes. Ein Wert Null zeigt an, da. das Subsystem keinen Wert f.r eine Fehlerbedingung zuweist.

. **subSystem**

Enth.lt eine Zeichenkette mit dem Namen des Subsystems, welches den Fehleraufruf generiert hat. F.r Fehler mit Standard Clipper-Operatoren und -Funktionen liefert diese Variable den Namen "BASE". F.r Fehler, die in einem Datenbanktreiber erzeugt wurden, enth.lt Error:subSystem den Namen des Datenbanktreibers.

. tries

Enth.lt einen ganzzahligen numerischen Wert, der die Anzahl der Wiederholungsversuche angibt. Wenn Error:canRetry auf true (.T.) gesetzt ist, kann dieser Wert dazu benutzt werden, die Anzahl der fehlerhaften Wiederholungen zu begrenzen.

Die Klassen-Funktion

Export-Instanzvariablen

Export Methoden

Methoden zur Status.nderung

Methoden zur Cursorsteuerung

Methoden zur Editierung

Methoden zur Texteingabe

Get-Klasse
  Bereitstellung von Objekten zum interaktiven Editieren von
  Datenbankfeldern und Variablen

Beschreibung
  Ein GET-Objekt ist ein grunds.tzlich unterst.tzter Mechanismus zum
  Editieren von Datenfeldern. Er wird in Clipper 5.0 ben.tigt, um @...GET-
  und READ-Befehle zu implementieren. GET-Objekte stellen ein
  ausgekl.geltes System zur Formatierung und Editierung von Daten
  einschlie.lich Cursorsteuerung und Daten.berpr.fung bereit. Die
  Daten.berpr.fung wird durch vom Anwender angegebene Code-Bl.cke
  erm.glicht und die Anzeigeformatierung durch die Verwendung von
  standardisierten Eingabemasken.

  Normalerweise verweist ein GET-Objekt auf eine einzelne Variable (in
  diesem Kapitel als Get-Variable bezeichnet). Das GET-Objekt greift nicht
  direkt auf die Variable zu, stattdessen wird die Variable durch
  Ausf.hrung eines unterst.tzenden Code-Blocks manipuliert. Wenn ein
  GET-Objekt .ber einen Standardbefehl @...GET erzeugt wird, wird
  automatisch ein Code-Block angelegt, der den Zugriff auf die in der
  Befehlszeile benannte Variable steuert. F.r ein GET-Objekt, erzeugt
  durch die Methode GetNew(), mu. ein passender CodeBlock angegeben
  werden.

  Ein GET-Objekt erh.lt den Wert einer GET-Variablen durch Berechnung
  eines unterst.tzenden Code-Blockes. Der Wert wird in eine darstellbare
  Form konvertiert, und in einen Puffer kopiert. Dieser Puffer kann vor
  dem Editieren auf dem Bildschirm angezeigt werden. Programmierbare
  Verhaltensweisen erm.glichen die Steuerung des Puffers und somit auch
  das Zur.ckkopieren des Inhaltes in die GET-Variable.

Beispiele
  . Dieses Beispiel erzeugt ein neues Objekt, weist den Instanzvariablen
    einige neue Attribute zu und editiert das GET mit der Funktion
    READMODAL():

```
LOCAL cVar := SPACE(10)
//
// Ein neues Objekt erzeugen
objGet := GetNew()
//
// Zuweisung einiger Instanzvariablen
objGet:row := 10
objGet:col := 2
//

// Zuweisung des Variablennamens und des Blockes.
objGet:name := "cVar"
objGet:block := {|cWert| IF(PCOUNT() > 0, cVar := cWert, cVar) }
//
objGet:picture := "@!"
objGet:colorSpec := "+BG/B, +W/BG"
objGet:postBlock := {|cWert| .NOT. EMPTY(cWert) }
//
// Editieren eines einzelnen GET-Objektes
READMODAL({objGet})
```

Dateien
  Verweis auf die Quelldatei ist Getsys.prg

See Also:

Syntax

```
GetNew([<nZeile>], [<nSpalte>], [<bBlock>],
       [<cVarName>],[<cMaske>], [<cFarben>]) -> objGet
```

Anwendung

GetNew() liefert ein neues GET-Objekt mit Instanzvariablen f.r Zeile, Spalte, Block, Maske und Farbangaben, deren Werte .ber Parameter angegeben sind.

badDate

Enth.lt einen logischen Wert, der anzeigt, ob der Puffer einen ung.ltigen Datumswert enth.lt. Get:badDate liefert true (.T.), wenn eine GET-Variable vom Typ Datum und der im Puffer befindliche Wert ung.ltig sind. Ein Wert false (.F.) in Get:badDate bedeutet, da. entweder ein g.ltiger Datumswert vorliegt oder die GET-Variable kein Datums-typ ist.

. block (zuweisungsf.hig)

Enth.lt einen Code-Block, der benutzt wird, um eine Variable mit einem GET-Objekt zu verbinden. Der Code-Block nimmt ein optionales Argument entgegen, dessen Wert der Variablen zugewiesen werden sollte. Wenn kein Parameter angegeben ist, gibt der Code-Block den aktuellen Wert der Variablen zur.ck.

Hinweis
Benutzen Sie die vorzuziehenden Methoden varGet() und varPut() um auf Get-Variablen zuzugreifen.

See Also:

. **buffer (zuweisungsf.hig)**

Enth.lt eine Zeichenkette, die als Puffer zum Editieren des GET-Objektes bestimmt ist. Get:buffer ist nur dann von Bedeutung, wenn die Eingabe eines GET-Objektes aktiv ist. In allen anderen F.llen enth.lt der Puffer NIL und jeder Versuch einer Zuweisung wird ignoriert.

. cargo (zuweisungf.hig)

Die Cargo-Information kann einen Wert beliebigen Typs aufnehmen. Dieser Wert wird von dem Objekt selber nicht benutzt, er steht vielmehr dem Programmierer als eine Art Huckepack-Information zur Verf.gung, auf die jederzeit aus einem Programm heraus zugegriffen werden kann.

. **changed**

Enth.lt einen logischen Wert, der besagt, da. der Get:buffer seit dem Abschlu. der letzten Eingabe ver.ndert wurde. Get:changed enth.lt den Wert true (.T.), wenn der Puffer ge.ndert wurde, ansonsten enth.lt Get:changed den Wert false (.F.).

. col (zuweisungf.hig)

Enth.lt einen numerischen Wert, der die Bildschirmspalte angibt, ab der das GET-Feld am Bildschirm angezeigt wird.

. **colorSpec (zuweisungf.hig)**

Enth.lt eine Zeichenkette zur Angabe der Farbdarstellung eines GET-Objektes. Die Zeichenkette mu. zwei Farbangaben enthalten. Die erste Angabe entspricht dem Farbattribut f.r ein nicht aktives GET-Objekt (unselected). Die zweite Angabe bezieht sich auf das Farbattribut eines aktiven GET-Objektes (selected).

Werden keine Farbdefinitionen angegeben, so werden die Farben, die zuletzt .ber SETCOLOR() f.r enhanced und unselected gesetzt wurden, eingesetzt.

Informationen .ber Farbangaben und SETCOLOR() entnehmen Sie bitte der Funktionsbeschreibung im Kapitel Funktionen.

. **decPos**

Enth.lt einen numerischen Wert, der die Position des Dezimalpunktes innerhalb des Puffers angibt. Dieser Wert ist nur dann von Bedeutung, wenn ein numerisches Feld editiert wird und die Eingabe aktiv ist. Andernfalls enth.lt Get:decPos den Wert NIL.

. **hasFocus**

Enth.lt einen logischen Wert, der besagt, da. eine Eingabe aktiv ist.
Get:hasFocus enth.lt den Wert true (.T.), wenn die Eingabe aktiv ist,
ansonsten den Wert false (.F.)

. name (zuweisungsf.hig)

Enth.lt eine Zeichenkette mit dem Namen der GET-Variablen. Dieser Wert ist optional und kann f.r eigene Zwecke verwendet werden. Ein GET-Objekt, das durch einen @...GET-Befehl erzeugt wurde, enth.lt in Get:name den in diesem Befehl angegebenen Variablen-Namen.

Das Get-Objekt selber ignoriert diese Variable. Sie wird im Standardbefehl READ und in der Funktion READMODAL() benutzt, um die Funktion READVAR() zu versorgen (Sehen Sie auch die Funktionsbeschreibung READVAR()).

. original

Enth.lt den Wert eines bestimmten Datentyps und ist eine Kopie der GET-Variablen, bevor sie editiert wird. Der Wert wird von Get:undo() ben.tigt, um den Originalwert herzustellen. Get:Original ist nur dann von Bedeutung, wenn eine Eingabe aktiv ist, ansonsten ist der Wert NIL.

. **picture (zuweisungsf.hig)**

Enth.lt eine Zeichenkette, die als Eingabemaske zur Formatierung und Eingabe eines GET-Objektes verwendet wird. Die Maskensymbole und -funktionen werden in der Befehlsbeschreibung @...GET beschrieben.

. pos

Enth.lt einen numerischen Wert, der die Cursorposition innerhalb des Eingabefeldes angibt. Get:pos ist nur dann von Bedeutung, wenn eine Eingabe aktiv ist. In allen anderen F.llen ist der Wert NIL enthalten.

. **postBlock (zuweisungsf.hig)**

Enth.lt einen optionalen Code-Block, der verwendet wird, um eine G.ltigkeits.berpr.fung des neu eingegebenen Wertes durchzuf.hren. Wird er angegeben, mu. er einen Ausdruck enthalten, der den Wert true (.T.) zur.ckgibt, wenn der neue Wert g.ltig ist, oder false (.F.), wenn der Wert ung.ltig ist. Bei GET-Objekten, die .ber den Stan-dardbefehl @...Get erzeugt wurden, wird der Ausdruck der VALID-Bedingung dieser Instanz-variablen zugewiesen. Das GET-Objekt selber ignoriert diese Variable. Sie wird im Standardbefehl READ benutzt, um die VALID-Bedingung eines GET-Feldes zu implementieren. Im Standardbefehl READ wird Get:postBlock der Wert eines GETs als Parameter .bergeben, wenn dieser ausgef.hrt wird. Dieser Parameter gestattet dem Get:postBlock die .berpr.fung durch den direkten Vergleich mit der GET-Variablen.

BREAK() Verzweigt hinter das Ende eines BEGIN SEQUENCE...END

Syntax

BREAK(<exp>) -> NIL

Parameter

<exp> ist ein Wert, der dem n.chsten RECOVER-Statement .bergeben wird, falls ein solches angegeben ist. Beachten Sie, da. der Parameter angegeben werden mu. (er ist nicht optional).

R.ckgabe

BREAK() besitzt keinen R.ckgabewert.

Anwendung

Die BREAK()-Funktion ist identisch mit der BREAK-Anweisung einer BEGIN SEQUENCE...END Kontrollstruktur. Die Funktion darf nur innerhalb eines solchen Anweisungsblockes stehen.

Beispiele

Das folgende Beispiel zeigt, wie eine Sequence-Struktur mit einem Code-Block abgebrochen wird:

```
bSave := ERRORBLOCK({ |x| BREAK(x) } )

BEGIN SEQUENCE
  .
  .
  .
RECOVER USING objError
  .
  .
  .
END

ERRORBLOCK(bSave)
```

See Also:

. **preBlock (zuweisungsf.hig)**

Enth.lt einen optionalen Code-Block, der entscheidet, ob editiert werden darf. Ist er angegeben, so wird er ausgef.hrt und mu. true (.T.) zur.ckgeben, wenn die Eingabe erlaubt ist, ansonsten false (.F.). In GET-Objekten, die .ber den Standardbefehl @...Get...WHEN erzeugt wurden, wird Get:preBlock der Ausdruck zugewiesen, der durch die WHEN-Bedingung angegeben ist.

Das GET-Objekt selbst ignoriert diese Variable. Sie wird im Standard-befehl READ benutzt, um die WHEN-Bedingung zu implementieren.

. **reject**

Enth.lt einen logischen Wert, der anzeigt, ob das letzte Zeichen .ber Get:insert() oder Get:overStrike in den Puffer gestellt wurde. Get:reject enth.lt true (.T.), wenn das letzte Zeichen verlorengeht, ansonsten false (.F.). Beachten Sie, da. weitere Eingaben den Wert zur.cksetzen k.nnen.

. row (zuweisungsf.hig)

Enth.lt einen numerischen Wert, der die Zeile des angezeigten GET-Feldes angibt.

. **type**

Enth.lt ein einzelnes Zeichen, das den Typ der GET-Variablen angibt.
Weitere Informationen .ber die Darstellung der Datentypen k.nnen Sie der
Funktionsbeschreibung VALTYPE() im Kapitel Funktionen entnehmen.

. **typeOut**

Enth.lt einen logischen Wert, der anzeigt, ob mit der letzten Nachricht versucht wurde, die Position des Cursors au.erhalb des Pufferbereiches zu bewegen, oder ob die Position im Puffer kein Editieren zul..t. Get:typeOut enth.lt den Wert true (.T.), wenn die Bedingung erf.llt ist. Beachten Sie bitte, da. jede Methode zur Cursorbewegung den Wert in Get:typeOut zur.cksetzt.

. **assign() -> self**

Weist den Wert des Puffers durch Ausf.hrung des Code-Blocks Get:Block der GET-Variablen zu, wenn der Code-Block den an ihn .bergebenen Parameter unterst.tzt. Diese Methode ist nur dann von Bedeutung, wenn die Eingabe aktiv ist (hasFocus).

. **display() -> self**

Zeigt ein Get-Feld am Bildschirm an. Ist die Eingabe eines GET-Objektes aktiv, wird der Inhalt des Puffers in der Farbe selected dargestellt und der Cursor auf dem Bildschirm angezeigt. Die Position des Cursors innerhalb des Eingabefeldes entspricht der aktuellen Eingabeposition innerhalb des Puffers. Ist eine Eingabe nicht aktiv, ist Get:block durchgef.hrt und das Ergebnis wird mit dem Farbattribut unselected angezeigt.

. **killFocus() -> self**

  Deaktiviert die Eingabe eines GET-Objektes. Nachdem das Objekt diese
  Nachricht erhalten hat, wird der Inhalt des Puffers neu angezeigt und
  interne Statusinformationen verworfen.

. **reset() -> self**

Setzt alle internen Statusinformationen eines GET-Objektes zur.ck. Das schlie.t das Zur.cksetzen des Puffers auf den aktuellen Wert der GET-Variable und die Neupositionierung des Cursors auf die erste editierbare Stelle des Puffers ein. Diese Nachricht ist nur von Bedeutung, wenn die Eingabe aktiv ist.

. **setFocus() -> self**

Aktiviert die Eingabe eines Objektes. Nachdem das Objekt diese Nachricht erhalten hat, werden interne Statusinformationen, einschlie.lich der Export-Instanzvariablen Get:buffer, Get:pos, Get:decPos und Get:original, erzeugt und initialisiert. Der Inhalt des Puffers wird mit dem Farbattribut selected angezeigt.

. undo() -> self

Setzt die GET-Variable auf den Wert zur.ck, den ein GET vor Aktivierung einer Eingabe besa.. Diese Nachricht ist nur von Bedeutung, wenn die Eingabe aktiv ist.

Wird der GET-Variablen der gespeicherte Wert von Get:original zugewiesen, und anschlie.end eine Nachricht Get:reset() gesendet, hat dies die gleichen Auswirkungen, wie das Senden der Nachricht Get:undo.

. **updateBuffer() -> self**

Setzt den Wert des Puffers auf den aktuellen Wert der GET-Variable zur.ck, und zeigt den Wert neu am Bildschirm an. Diese Nachricht ist nur dann von Bedeutung, wenn die Eingabe aktiv ist.

varGet() -> GetVariablenWert

Liefert den Wert einer aktuellen Get-Variable. Mit einer einfachen Get-Variable ist diese Methode identisch mit der Ausf.hrung des Code-Blocks Get:block.

varGet() == EVAL(Get:block)

Wenn die Getvariable ein Array-Element ist, kann der Wert der Variable nicht .ber EVAL(Get:block) ermittelt werden. Benutzen Sie stattdessen die Methode varGet(). Ein Beispiel finden Sie in GETSYS.PRG innerhalb der Funktion READMODAL().

See Also:

varPut(<xWert>) -> GetVariablenWert

Setzt eine aktuelle Get-Variable auf den .bergebenen Wert. Mit einer einfachen Get-Variable ist diese Methode identisch mit der Ausf.hrung des Code-Blocks Get:block mit einem Parameter.

varPut() == EVAL(Get:block, <xWert>)

Wenn die Getvariable ein Array-Element ist, kann der Wert der Variable nicht .ber EVAL(Get:block) gesetzt werden. Benutzen Sie stattdessen die Methode varPut(). Ein Beispiel finden Sie in GETSYS.PRG innerhalb der Funktion READMODAL().

See Also:

. end() -> self

Bewegt den Cursor zur rechten, .u.eren editierbaren Position innerhalb des Puffers.

. **home() -> self**

  Bewegt den Cursor zur linken, .u.eren editierbaren Position innerhalb
  des Puffers.

. **left() -> self**

  Bewegt den Cursor nach links zur n.chsten editierbaren Position
  innerhalb des Puffers. Ist links vom Cursor keine weitere editierbare
  Position vorhanden, so bleibt die Position des Cursors unbeeinflu.t.

. **right() -> self**

Bewegt den Cursor nach rechts zur n.chsten editierbaren Position innerhalb des Puffers. Ist rechts vom Cursor keine weitere editierbare Position vorhanden, bleibt die Position des Cursors unbeeinflu.t.

. **toDecPos() -> self**

Bewegt den Cursor direkt zur Position rechts vom Dezimalpunkt innerhalb des Puffers.

Diese Nachricht ist nur dann von Bedeutung, wenn ein numerischer Wert editiert wird.

. **wordLeft() -> self**

  Bewegt den Cursor nach links zum n.chsten Wort innerhalb des Puffers.
  Befindet sich der Cursor bereits auf dem ersten Wort, bleibt die
  Position des Cursors unbeeinflu.t.

. **wordRight() -> self**

  Bewegt den Cursor nach rechts zum Anfang des n.chsten Wortes innerhalb
  des Puffers. Befindet sich der Cursor bereits auf dem letzten Wort, bleibt
  die Position des Cursors unbeeinflu.t.

. **backspace() -> self**

L.scht das Zeichen links vom Cursor und bewegt den Cursor eine Stelle nach links.

Befindet sich der Cursor bereits in der .u.erst linken Position, bleibt die Nachricht ohne Bedeutung.

. **delLeft() -> self**
  L.scht ein Zeichen links vom Cursor.

. **delRight() -> self**
  L.scht ein Zeichen rechts vom Cursor.

. **delete() -> self**
  L.scht ein Zeichen an der Curorposition.

. **delWordLeft() -> self**
  L.scht ein Wort links vom Cursor.

. **delWordRight() -> self**
  L.scht ein Wort rechts vom Cursor.

. **insert(<cZeichen>) -> self**
F.gt ein Zeichen <cZeichen> an der aktuellen Position des Cursors in den Puffer ein und schiebt den restlichen Inhalt ab der Cursorposition nach rechts. Der Cursor wird
anschlie.end eine Stelle nach rechts plaziert.

. **overStrike(<cZeichen>) -> self**
  Setzt ein Zeichen <cZeichen> an die aktuelle Position des Cursors
  innerhalb des Puffers, .berschreibt ein existierendes Zeichen und bewegt
  den Cursor eine Stelle nach rechts.

Klassen-Funktionen

Export-Instanzvariablen

TBColumn-Klasse
  Bereitstellung von Spalten-Objekten f.r TBrowse-Objekte

Beschreibung

  Ein TBColumn-Objekt ist ein einfaches Objekt mit den ben.tigten
  Informationen f.r eine einzelne Spalte in einem TBrowse-Objekt.
  TBColumn-Objekte enthalten keine Methoden, lediglich
  Export-Instanzvariablen.

Beispiele

  . Das Beispiel zeigt einen Programmteil, in dem ein TBrowse-Objekt
    angelegt wird und einige TBColumn-Objekte hinzugef.gt werden:

```
USE Kunden NEW
//
// Anlegen eines neue TBrowse-Ojektes
objBrowse := TBrowseDB(1, 0, 23, 79)
//
// Einige Spalten anlegen und hinzuf.gen
objBrowse:addColumn(TBColumnNew( "Kunde", {|| Kunden.Name } ))
objBrowse:addColumn(TBColumnNew( "Ort", {|| Kunden.Ort } ))
objBrowse:addColumn(TBColumnNew( "Telefon", {|| Kunden.Telefon } ))
.
.<Anweisungen, um die Daten zu browsen>
.
CLOSE Kunden
```

  . Ein vollst.ndig anwendbares Beispiel mit TBrowse-Objekten finden Sie
    im Verzeichnis \CLIPPER5\SOURCE\SAMPLE in der Datei TbDemo.prg.

See Also:

TBColumnNew(<cKopf>,<bBlock>) -> objTBColumn

Liefert ein neues TBColumn-Objekt mit angegebenem Spaltenkopf und einem Block zur Ermittlung der Daten. Weitere Elemente des Objektes k.nnen direkt durch Zuweisung der Instanzvariablen mit Werten versehen werden.

. block (zuweisungsf.hig)

Enth.lt einen Code-Block, der die Datenwerte der Spalten entgegennimmt. Jeder Code-Block ist g.ltig und es werden keine Parameter an den Code-Block .bergeben, wenn dieser ausgef.hrt wird. Der Code-Block mu. den, f.r diese Datenspalte g.ltigen Datentyp liefern.

. **cargo (zuweisungsf.hig)**

Die Cargo-Information kann einen Wert beliebigen Typs aufnehmen. Dieser Wert wird von dem Objekt selber nicht benutzt, er steht vielmehr dem Programmierer als eine Art Huckepack-Information zur Verf.gung, auf die jederzeit aus einem Programm heraus zugegriffen werden kann.

. **colorBlock (zuweisungsf.hig)**

Enth.lt optional einen Code-Block, der Farben beschreibt, mit denen ein Datenwert am Bildschirm angezeigt wird. Der Code-Block wird, wenn angegeben, jedesmal dann ausgef.hrt, wenn .ber TBColumn:block neue Werte entgegengenommen werden. Dem Code-Block TBColumn:colorBlock wird der neue Datenwert als Parameter .bergeben und er mu. als Ergebnis ein Array mit zwei numerischen Elementen zur.ckliefern. Die R.ckgabewerte werden als Index auf die Farbtabelle in TBrowse:colorSpec benutzt, um Farbattribute f.r standard und enhanced zu erhalten (wird in TBColumn:defColor n.her beschrieben).

TBColumn:colorBlock erlaubt die unterschiedliche Farbgestaltung einer Datenanzeige in Abh.ngigkeit von einem Wert. So k.nnen zum Beispiel positive und negative Datenwerte verschiedenfarbig angezeigt werden.

. **colSep (zuweisungsf.hig)**

Enth.lt optional eine Zeichenkette zum Trennen der einzelnen Datenspalten. Die hier angegebenen Zeichen werden an der linken Seite der jeweiligen Spalte zur Anzeige gebracht, wenn dort eine weitere Spalte vorhanden ist. Ist kein Wert in dieser Instanzvariablen angegeben, wird stattdessen der Wert in TBrowse:colSep angezeigt.

. **defColor (zuweisungsf.hig)**

Enth.lt ein Array mit zwei numerischen Elementen, die als Index auf die .ber TBrowse:colorSpec angegebene Farbtabelle dienen. Der Wert des ersten Elementes zeigt innerhalb der Farbtabelle auf Farbattribute zur Darstellung der Kopfzeilen, Fu.zeilen und auf Daten, die nicht durch den Browse-Cursor selektiert sind.

Der zweite Wert zeigt innerhalb der Farbtabelle auf das Farbattribut, das zur Darstellung der durch den Cursor selektierten Datenzeile verwendet wird.

Der Vorgabewert ist ein Array mit den numerischen Werten 1 und 2 ({1,2}). Dies bedeutet, da. die ersten zwei Angaben in der Farbtabelle TBrowse:colorSpec zur Darstellung von standard und enhanced benutzt werden.
Beachten Sie, da. ein Code-Block in TBColumn:colorBlock die Werte in TBcolumn:defColor .berschreibt.

. footing (zuweisungsf.hig)

Enth.lt eine Zeichenkette zur Darstellung von Fu.zeilen f.r diese Spalte.

. **footSep (zuweisungsf.hig)**

Enth.lt eine Zeichenkette zur Trennung von Daten- und Fu.bereich f.r diese Spalte.

. heading (zuweisungsf.hig)

Enth.lt eine Zeichenkette zur Darstellung von Kopfzeilen f.r diese Spalte.

. headSep (zuweisungsf.hig)

Enth.lt eine Zeichenkette zur Trennung von Daten- und Kopfbereich f.r diese Spalte.

. width (zuweisungsf.hig)

Enth.lt als numerischen Wert die angezeigte Breite einer Spalte. Ist kein Wert angegeben, so wird die Breite .ber den Wert, der von TBColumn:block zur.ckgegeben wird, berechnet.

Optionale Match-Klauseln k.nnen an beliebiger Stelle und in beliebiger Reihenfolge im Austausch-Muster vereinbart werden. Jede optionale Match-Klausel darf im Quelltext nur einmal vorkommen. Es existieren zwei Arten von Optional-Klauseln: die erste wird durch ein Schl.sselwort gefolgt von einem Match-Marker angegeben, die zweite enth.lt nur einen Match-Marker. Mit diesen beiden Typen optionaler Vereinbarungen k.nnen alle traditionellen Befehlsvereinbarungen des Clipper-Befehlsumfanges abgedeckt werden.

Optionale Klauseln werden in Kombination mit einem Regular- oder List-Marker definiert, um Quelltext entgegenzunehmen, wenn die Vereinbarung ein Schl.sselwort gefolgt von einem Argument, oder nur ein Argument enth.lt. Ein Beispiel hierf.r ist die INDEX-Klausel eines USE-Befehls. W.rde diese optionale Vereinbarung lediglich aus dem Schl.sselwort INDEX bestehen, k.nnte sie durch einen Restricted-Marker entgegengenommen werden. Da jedoch ein Argument erforderlich ist, mu. in dieser optionalen Vereinbarung zus.tzlich ein List-Match-Marker angegeben werden. Ein Beispiel f.r die Verwendung eines optionalen Restricted-Match-Markers ist dagegen die im USE-Befehl optional enthaltene EXCLUSIV- oder SHARED-Klausel.

Werden innerhalb eines Austauschmusters mehrere optionale Match-Klauseln mit gleichen Schl.sselworten unmittelbar hintereinander jeweils in eckigen ([]) Klammern angegeben, so f.hrt dies zu einem Compilier-Fehler. Stehen solche Definitionen jedoch nicht unmittelbar hintereinander oder sind die Schl.sselworte unterschiedlich, d.rfen sie beliebig oft angegeben werden. Um wiederholte Match-Klauseln in den Ergebnistext zu .bernehmen, k.nnen auch mehrfache Result-Klauseln im Ergebnistext angegeben werden. Lesen Sie dazu bitte den folgenden Abschnitt.

BROWSE() Datens.tze in einem Fenster browsen

Syntax

BROWSE([<nOben>], [<nLinks>],
[<nUnten>], [<nRechts>]) -> NIL

Parameter

<nOben>, <nLinks>, <nUnten> und <nRechts> definieren Fensterkoordinaten, innerhalb derer die Datens.tze angezeigt werden sollen. Wird nichts angegeben, so sind die voreingestellten Koordinaten 1, 0 bis MAXROW(), MAXCOL().

R.ckgabe

BROWSE besitzt keinen R.ckgabewert.

Anwendung

BROWSE ist eine anwenderorientierte Funktion und wird haupts.chlich zur tabellenorientierten Darstellung und als Editor f.r Datens.tze im aktuellen Arbeitsbereich angewendet. Eine Liste der Tastenfunktionen zur Bedienung von BROWSE() ist in der Funktion DBEDIT() beschrieben.

. Statuszeile: BROWSE() zeigt eine Statuszeile in der oberen rechten Ecke des definierten Fensters an, welche folgende Bedeutungen enthalten:

Tabelle 5-6: BROWSE() Statusmeldungen

Meldung	Bedeutung
<neu>	Anf.ge-Modus (APPEND)
<bot>	Anfang der Datei
<gel.scht>	Aktueller Datensatz ist gel.scht
Satz Nr.	Anzeige der aktuellen Datensatznummer

. Modi: BROWSE() besitzt drei verschiedene Zust.nde: browsen, Feld editieren und neuen Datensatz anf.gen.

Browsen: Dies ist der voreingestellte Modus von BROWSE(). Durch bet.tigen der DBEDIT()-Bedientasten wird der hervorgehobene Browsezeiger zu einer neuen Zeile oder Spalte bewegt.

Feld editieren: Die Bet.tigung der Taste [Return] in einem beliebigen Feld best.tigt das Editieren eines Feldes mit GET. Nochmalige Bet.tigung von [Return] beendet das Editieren und speichert die .nderung.
[Esc] ric
Befindet sich das Feld im Edit-Modus, gelten zur Editierung die Tastenfunktionen von READ.

Anf.gen: Die Bet.tigung der Tasten [Strg]+[Bild.] und anschlie.endes Dr.cken der Taste [.] wechselt in den Anf.ge-Modus. Dies wird durch die Meldung <neu> in der Statuszeile angezeigt. Ein neuer, leerer Datensatz wird an das Ende der Datei angef.gt. Durch Bet.tigen der Taste [.] wird der Anf.ge-Modus verlassen und der neue Datensatz gesichert, wenn Daten eingegeben worden sind. Wurden keine neuen Daten eingegeben, wird der Datensatz gel.scht und nicht in der aktuellen Datei gespeichert.

Dateien

Bibliothek ist EXTEND.LIB, Quelldatei ist Examplep.prg

See Also:

Klassen-Funktionen

Export-Instanzvariablen

Export Methoden

Cursorbewegungs-Methoden

Sonstige Methoden

TBrowse-Klasse

Bereitstellung von Objekten f.r tabellenorientiertes Anzeigen von Daten (BROWSE)

Beschreibung

Ein TBrowse-Objekt ist ein generell zur Verf.gung gestellter Mechanismus zum Browsen tabellenorientierter Daten. TBrowse-Objekte stellen eine anspruchsvolle Behandlungsweise zur Abfrage, Formatierung und Anzeige von Daten dar. Datenentgegennahme und Dateipositionierung werden .ber anwenderdefinierte Code-Bl.cke durchgef.hrt und erm.glichen dadurch einen hohen Grad an Flexibilit.t und Interaktion zwischen dem Browse-Mechanismus und und den unterliegenden Quelldaten. Das Format der individuellen Datenwerte kann pr.zise .ber Daten entgegennehmende Code-Bl.cke kontrolliert werden. Alle Darstellungen, Formatierungen und Farbattribute k.nnen durch Senden von Nachrichten an ein TBrowse-Objekt gesteuert und kontrolliert werden.

Ein TBrowse-Objekt ben.tigt eine oder mehrere TBColumn-Objekte. Ein TBColumn-Objekt enth.lt die ben.tigten Informationen, um jeweils eine Spalte einer Browse-Tabelle zu definieren (Siehe auch unter TBColumn-Klasse in diesem Kapitel).

Wird eine Operation durchgef.hrt, nimmt ein TBrowse-Objekt die Daten durch Ausf.hrung eines Code-Blockes entgegen. Die Daten sind innerhalb eines Bildschirmbereiches (Browse-Fenster) in Zeilen und Spalten organisiert. Ein TBrowse-Objekt besitzt einen eigenen Browse-Cursor. Das Datenfeld, auf den der Browse-Cursor zeigt, wird hervorgehoben dargestellt (Der aktuelle Bildschirmcursor wird innerhalb des Datenfeldes in der ersten Position angezeigt).

Bei der Initialisierung wird der Browse-Cursor auf das Datenfeld in der obersten linken Ecke des Browse-Fensters gesetzt. Es k.nnen Nachrichten an das Objekt gesendet werden, um den Browse-Cursor innerhalb der Tabelle zu positionieren. Diese Nachrichten werden normalerweise als Reaktion eines vom Benutzer ausgel.sten Tastendruckes gesendet.

Neue darzustellende Daten werden automatisch abgeholt, wenn die Bewegung des Browse-Cursors innerhalb der Tabelle es erfordert. Eine versuchte Cursor-Bewegung .ber die Grenzen des Browse-Fensters hinaus veranla.t, da. verdeckte Spalten oder Zeilen automatisch in den sichbaren Bereich gebracht werden. Die zugrundeliegenden Quelldaten werden dabei automatisch .ber die Ausf.hrung eines Code-Blockes repositioniert und in diese neuen Spalten oder Zeilen .bernommen.

Beispiele

Ein vollst.ndig anwendbares Beispiel mit TBrowse-Objekten finden Sie im Verzeichnis \CLIPPER5\SOURCE\SAMPLE in der Datei TbDemo.prg.

See Also:

TBrowseNew(<nOben>, <nLinks>, <nUnten>, <nRechts>) -> objTBrowse

Liefert ein neues TBrowse-Objekt mit den angegebenen Bildschirmkoordinaten.
Das TBrowse-Objekt wird ohne Spalten (TBColumns) und ohne Code-Bl.cke zur
Positionierung von Quelldaten angelegt. Diese Angaben m.ssen erfolgen,
bevor das TBrowse-Objekt benutzt wird.

TBrowseDB(<nOben>, <nLinks>, <nUnten>, <nRechts>) -> objTBrowse

Liefert ein neues TBrowse-Objekt mit angegebenen Bildschirmkoordinaten
und vorbelegten Code-Bl.cken f.r die Positionierung von Quelldaten einer
Datenbank-Datei. Die vorbelegten Code-Bl.cke sind f.r die Ausf.hrung der
Operationen GO TOP, GO BOTTOM und Skip zust.ndig.
Beachten Sie, da. TBrowseDB ein Objekt ohne Spalten (TBColumn) anlegt.
Bevor ein TBrowse-Objekt angewendet wird, m.ssen Spalten zur Darstellung
jedes Feldes hinzugef.gt werden.

. **autoLite (zuweisungsf.hig)**

Enth.lt einen logischen Wert. Wenn Get:autoLite auf true (.T.) gesetzt wird, setzt die Methode Get:stabilize() automatisch die aktuelle Zelle auf hervorgehobene Darstellung.

. **cargo (zuweisungsf.hig)**

Die Cargo-Information kann einen Wert beliebigen Typs aufnehmen. Dieser Wert wird von dem Objekt selber nicht benutzt, er steht vielmehr dem Programmierer als eine Art Huckepack-Information zur Verf.gung, auf die jederzeit aus einem Programm heraus zugegriffen werden kann.

. **colCount**

Enth.lt einen numerischen Wert, der die Anzahl der Datenspalten angibt.
F.r jede Spalte ist eine entsprechendes TBColumn-Objekt vorhanden.

. colorSpec (zuweisungsf.hig)

Enth.lt eine Zeichenkette mit einer Farbtabelle zur Darstellung der TBrowse-Objekte. Als Vorgabewerte werden die aktuellen Einstellungen von SETCOLOR() in diese Instanzvariable kopiert, wenn das Objekt angelegt wird. Informationen .ber die Farbcodes entnehmen Sie bitte der Funktions-beschreibung zu SETCOLOR().

See Also:

. colPos (zuweisungsf.hig)

Enth.lt einen numerischen Wert, der die Position der aktuellen Datenspalte enth.lt. Die Z.hlweise beginnt bei Null und in der linken, .u.eren Datenspalte.

. colSep (zuweisungsf.hig)

Enth.lt eine Zeichenkette, die zur Trennung der Spalten angezeigt wird, wenn ein TBColumn-Objekt keinen eigenen Spaltentrenner enth.lt (Siehe auch unter TBColumn-Klasse in diesem Kapitel).

. **freeze (zuweisungsf.hig)**

Enth.lt einen numerischen Wert, der besagt, wie viele Datenspalten, beginnend bei der linken Datenspalte, festgesetzt werden. Festgesetzte Spalten sind immer sichtbar, auch wenn andere Spalten am Bildschirm seitlich gerollt werden (panning).

. **goBottomBlock (zuweisungsf.hig)**

Kann einen Code-Block enthalten, der ausgef.hrt wird, wenn das TBrowse-Objekt die Nachricht TBrowse:goBottom() erh.lt. Dieser Code-Block wird ben.tigt, um die Quelldaten der letzten darzustellenden Datens.tze zu repositionieren. Normalerweise kommen die Quelldaten aus einer Datenbankdatei und der Code-Block enth.lt einen Aufruf einer anwenderdefinierten Funktion (UDF), die einen GO BOTTOM-Befehl ausf.hrt.

. goTopBlock (zuweisungsf.hig)

Kann einen Code-Block enthalten, der ausgef.hrt wird, wenn das TBrowse-Objekt die Nachricht TBrowse:goTop() erh.lt. Dieser Code-Block wird ben.tigt, um die Quelldaten der ersten darzustellenden Datens.tze zu repositionieren. Normalerweise kommen die Quelldaten aus einer Datenbankdatei und der Code-Block enth.lt einen Aufruf einer anwenderdefinierten Funktion (UDF), die einen GO TOP-Befehl ausf.hrt.

. **headSep (zuweisungsf.hig)**

Kann eine Zeichenkette als Trennung der Kopfzeilen von den Datenzeilen enthalten, die verwendet wird, wenn das TBColumn-Objekt keine eigene Kopftrennung enth.lt (Siehe auch unter TBColumn-Klasse in diesem Kapitel).

. **hitBottom (zuweisungsf.hig)**

Enth.lt einen logischen Wert, der anzeigt, da. versucht worden ist, hinter das Ende der verf.gbaren Daten zu positionieren.

TBrowse:hitBottom enth.lt true (.T.), wenn ein Versuch gemacht wurde, anderenfalls false (.F.). W.hrend einer Stabilisation setzt das TBrowse-Objekt diesen Wert auf true (.T.), wenn TBrowse:skipBlock anzeigt, da. eine Skip-Vorw.rts-Operation unm.glich ist.

. **hitTop (zuweisungsf.hig)**

Enth.lt einen logischen Wert, der anzeigt, da. versucht worden ist, vor den Anfang der verf.gbaren Daten zu positionieren. TBrowse:hitTop enth.lt true (.T.), wenn ein Versuch gemacht wurde, anderenfalls false (.F.). W.hrend einer Stabilisation setzt das TBrowse-Objekt diesen Wert auf true (.T.), wenn TBrowse:skipBlock anzeigt, da. eine Skip-R.ckw.rts-Operation unm.glich ist.

. **nBottom (zuweisungsf.hig)**

Enth.lt einen numerischen Wert, der die untere Bildschirmzeile des TBrowse-Bildschirmfensters angibt.

. nLeft (zuweisungsf.hig)

Enth.lt einen numerischen Wert, der die linke Bildschirmspalte des TBrowse-Bildschirmfensters angibt.

. nRight (zuweisungsf.hig)

Enth.lt einen numerischen Wert, der die rechte Bildschirmspalte des TBrowse-Bildschirmfensters angibt.

. nTop (zuweisungsf.hig)

Enth.lt einen numerischen Wert, der die obere Bildschirmzeile des TBrowse-Bildschirmfensters angibt.

. **rowCount**

Enth.lt einen numerischen Wert, der anzeigt, wieviele Bildschirmzeilen f.r die Darstellung der Daten im Bildschirmfenster zur Verf.gung stehen. Lediglich Datenzeilen sind hier angegeben, die Zeilen f.r Kopf- und Fu.bereich sowie f.r die Trennungen sind nicht eingeschlossen.

. **rowPos (zuweisungsf.hig)**

Enth.lt einen numerischen Wert, der die aktuelle Datenzeile angibt, in der sich der Cursor befindet. Datenzeilen beginnen bei eins, ab der obersten Zeile. Bildschirmzeilen f.r Kopf- und Fu.bereich sowie f.r die Trennungen sind davon ausgeschlossen.

. skipBlock (zuweisungsf.hig)

Enth.lt einen Code-Block, der zur Repositionierung der Quelldaten ben.tigt wird. W.hrend der Stabilisation wird dieser Code-Block ausgef.hrt und bekommt einen numerischen Parameter .bergeben. Der Parameter enth.lt die relative Satzpositionierung (skipping), ausgehend vom aktuellen Datensatzzeiger. Ein positiver Wert bedeutet eine Vorw.rtspositionierung (skip forward), ein negativer Wert bedeutet eine R.ckw.rtspositionierung (skip backward). Ein Wert Null bedeutet, da. keine Neupositionierung erfolgen soll, lediglich die Quelldaten des Datensatzes sollen aktualisiert (refreshed) werden.

Normalerweise kommen die Quelldaten aus einer Datenbankdatei und der Code-Block enth.lt einen Aufruf einer anwenderdefinierten Funktion (UDF). Diese soll einen SKIP-Befehl ausf.hren, um die Neupositionierung des Datensatzzeigers vorzunehmen.

Der Code-Block mu. die Anzahl der tats.chlich .bersprungenen Datens.tze zur.ckgeben (positiv, negativ oder Null). Weicht der R.ckgabewert von dem, .ber den Parameter angeforderten Wert ab, geht TBrowse davon aus, da. der Anfang oder das Ende der Datei erreicht ist.

. stable (zuweisungsf.hig)

Enth.lt einen logischen Wert, der anzeigt, da. sich TBrowse in einem stabilen Zustand befindet. Der Wert true (.T.) bedeutet stabiler und der Wert false (.F.) instabiler Zustand. Das TBrowse-Objekt befindet sich in einem stabilen Zustand, wenn alle Daten verarbeitet und angezeigt sind, der Datensatzzeiger mit der durch die Cursorposition bestimmten Datenzeile .bereinstimmt und die jeweilige Datenzeile hervorgehoben dargestellt wird. Wenn dem TBrowse-Objekt Nachrichten zur Neupositionierung gesendet werden (navigation messages), wird TBrowse:stable auf false (.F.) gesetzt. Nachdem eine Stabilisation .ber die Nachricht TBrowse:stabilize durchgef.hrt ist, wird der Wert in TBrowse:stable wieder auf true (.T.) gesetzt.

. down() -> self

Bewegt den Browse-Cursor eine Zeile abw.rts. Steht der Cursor bereits auf der untersten Datenzeile, wird der Datenbereich des Bildschirmes nach oben gerollt und in der untersten Datenzeile ein neuer Datensatz angezeigt. Steht der Cursor in der untersten Zeile und ist bereits das logische Ende der Datei erreicht, wird TBrowse:hitBottom auf true (.T.) gesetzt.

. end() -> self

Bewegt den Cursor in die .u.erste, rechte Datenspalte innerhalb des Browse-Fensters.

. **goBottom() -> self**

Neupositionierung der Quelldaten auf das Ende der Datenbank (EOF()), durch Ausf.hrung des Code-Blockes in TBrowse:goBottomBlock. Die Datenwerte der letzten, im Datenbereich des Browse-Fensters darstellbaren Datens.tze werden neu angezeigt und der Cursor wird in die unterste Datenzeile positioniert.

. **goTop() -> self**

Neupositionierung der Quelldaten auf den Anfang der Datenbank (BOF())
durch Ausf.hrung des Code-Blockes in TBrowse:goTopBlock. Die Datenwerte
der ersten, im Datenbereich des Browse-Fensters darstellbaren Datens.tze
werden neu angezeigt und der Cursor wird in die oberste Datenzeile
positioniert.

. home() -> self

Bewegt den Cursor in die .u.erste, linke und nicht festgestellte
(freezed) Datenspalte innerhalb des Browse-Fensters.

. left() -> self

Bewegt den Cursor eine Datenspalte weiter nach links. Befindet sich der Cursor bereits in der .u.ersten, linken Spalte, werden die nicht festgesetzten Spalten nach rechts gerollt und die vorherige Datenspalte (an der aktuellen Cursorposition) angezeigt.

. **pageDown() -> self**

Positioniert den Datenbereich eine Seite abw.rts und aktualisiert die Datenanzeige. Ist das logische Ende der Datei bereits erreicht, wird lediglich der Cursor in die unterste Datenzeile bewegt. Befindet sich dar.ber hinaus der Cursor bereits in der untersten Datenzeile, wird der Wert in TBrowse:hitBottom auf true (.T.) gesetzt.

. **pageUp() -> self**

Positioniert den Datenbereich eine Seite aufw.rts und aktualisiert die Datenanzeige. Ist der logische Anfang der Datei bereits erreicht, wird lediglich der Cursor in die oberste Datenzeile bewegt. Befindet sich dar.ber hinaus der Cursor bereits in der obersten Datenzeile, wird der Wert in TBrowse:hitTop auf true (.T.) gesetzt.

. **panEnd() -> self**

Bewegt den Cursor in die letzte Datenspalte. Wenn erforderlich, wird das Bildschirmfenster seitlich auf die letzte Datenspalte (TBColumn) gerollt.

. **panHome() -> self**

  Bewegt den Cursor in die erste Datenspalte. Wenn erforderlich, wird das
  Bildschirmfenster seitlich auf die erste Datenspalte (TBColumn) gerollt.

. **panLeft() -> self**

Rollt den Inhalt des Bildschirmfensters eine Datenspalte nach links, wenn m.glich, ohne die Cursorposition zu .ndern. Wenn mit TBrowse:panLeft() eine Datenspalte nach links gerollt wird, geht die Anzeige der linken Datenspalte verloren, da ein oder mehrere verf.gbare Datenspalten am rechten Rand neu angezeigt werden.

. **panRight() -> self**

Rollt den Inhalt des Bildschirmfensters eine Datenspalte nach rechts und wenn m.glich, ohne die Cursorposition zu .ndern. Wenn mit TBrowse:panRight() eine Datenspalte nach rechts gerollt wird, geht die Anzeige der rechten Datenspalte verloren, da ein oder mehrere,verf.gbare Datenspalten am linken Rand neu angezeigt werden.

. **right() -> self**

  Bewegt den Cursor eine Datenpalte weiter nach rechts. Befindet sich der
  Cursor bereits in der .u.ersten, rechten Spalte, werden die nicht
  festgesetzten Spalten nach links gerollt und die n.chste Datenspalte (an
  der aktuellen Cursorposition) angezeigt.

. up() -> self

Bewegt den Browse-Cursor eine Zeile aufw.rts. Steht der Cursor bereits auf der obersten Datenzeile, wird der Datenbereich des Bildschirms nach unten gerollt und in der obersten Datenzeile ein neuer Datensatz angezeigt. Steht der Cursor in der obersten
Zeile und ist bereits der logische Anfang der Datei erreicht, wird TBrowse:hitTop auf true (.T.) gesetzt.

. addColumn(<objColumn>) -> self

F.gt ein neues TBColumn-Objekt an ein TBrowse-Objekt an und erh.ht den Wert der Instanzvariablen TBrowse:colCount um eins.

. colorRect(<aBereich>, <aFarben>) -> self

Diese Methode wechselt direkt die Farbdarstellung eines Bereiches von Datenzellen. <aBereich> ist ein Array mit vier numerischen Werten, die den Datenbereich beschreiben. Die Angaben beziehen sich dabei auf Datenzellen und nicht auf einen Bildschirmbereich. <aFarben> ist ein Array mit zwei numerischen Werten. Die Werte werden als Index auf die Farbtabelle colorSpec verwendet und zeigen auf die Farbattribute f.r normale und hervorgehobene Darstellung.

Die mit colorRect() farblich markierten Zellen behalten Ihre Farben solange, bis sie aufw.rts oder abw.rts aus den Bildschimbereich gerollt werden. Horizontales Rollen (panning) hat keinen Einflu. auf die Farbdarstellung. Zellen, die sich links oder rechts au.erhalb des Bildschirmbereiches befinden, k.nnen ebenfalls farblich markiert werden.

Beispiel:

```
aRect := { 1, 1, browse:rowCount, browse:colCount }
browse:colorRect(aRect, {1, 2} )
```

. configure() -> self

Weist das TBrowse-Objekt an, alle Instanzvariablen und TBColumn-Objekte zu .berpr.fen und nimmt eine Rekonfiguration der erforderlichen internen Einstellungen vor. Diese Nachricht (Message) kann benutzt werden, wenn ein TBColumn-Objekt auf direktem Wege ge.ndert wurde.

. **deHilite() ->self**

  Schaltet die hervorgehobene Darstellung der aktuellen Zelle aus. Diese
  Methode wurde implementiert, um ein manuelle Hervorhebung zu erm.glichen,
  wenn autoHilite ausgeschaltet ist (auf false gesetzt).

. getColumn(<nSpalte>) -> objColumn

Liefert das im TBrowse-Objekt enthaltene und .ber <nSpalte> angegebene TBColumn-Objekt.

. **hilite() -> self**

  Schaltet die hervorgehobene Darstellung der aktuellen Zelle ein. Diese
  Methode wurde implementiert, um ein manuelle Hervorhebung zu erm.glichen,
  wenn autoHilite ausgeschaltet ist (auf false gesetzt).

. **refreshAll() -> self**

Durch diese Nachricht (Message) werden intern alle Datenzeilen als ung.ltig markiert. Dadurch werden bei der n.chsten Ausf.hrung von TBrowse:stabilize() die Datenwerte aktualisiert und neu angezeigt.

. **refreshCurrent() -> self**

  Markiert die aktuelle Datenzeile als ung.ltig. Dadurch werden Datenwerte
  dieser Zeile bei der n.chsten Ausf.hrung von TBrowse:stabilize()
  aktualisiert und neu angezeigt.

. **setColumn(<nSpalte>,<objColumn>) -> objColumn**

Ersetzt ein durch <nSpalte> spezifiziertes TBColumn-Objekt durch ein neues, .ber <objColumn> angegebenes Spaltenobjekt.

. stabilize() -> lStabil

F.hrt eine stufenweise Stabilisation des Datenbereiches durch. Immer dann, wenn diese Nachricht gesendet wird, wird ein Teil des Stabilisationsprozesses ausgef.hrt. Die stufenweise Durchf.hrung erlaubt eine Unterbrechung durch einen Tastendruck oder ein anderes Ereignis. Erst nachdem sich das ganze TBrowse-Objekt in einem stabilen Zustand befindet, wird true (.T.) zur.ckgegeben. Andernfalls wird false (.F.) zur.ckgegeben und es m.ssen weitere stabilize()-Nachrichten gesendet werden.
Das TBrowse-Objekt nimmt erst dann einen stabilen Zustand an, wenn alle Datenwerte aktualisiert und angezeigt sind, die Quelldaten mit dem entsprechenden Datensatz .bereinstimmen und die aktuelle Zelle hervorgehoben dargestellt ist.

.bersicht der Compiler-Warnungen (Nur bei /W)

.bersicht der Compiler-Fehlermeldungen

Schwerwiegende Compiler-Fehlermeldungen

Compiler-Fehlermeldungen-- Allgemein

Dieses Kapitel gibt eine .bersicht der Fehlermeldungen, die auftreten k.nnen, wenn ein Programm mit CLIPPER.EXE compiliert wird. Die Fehler sind entsprechend ihrer Schwere und der Stelle, an der sie in einem Compilations-Proze. auftreten k.nnen, in verschiedene Kategorien unterteilt. Im folgenden wird jede dieser Kategorien zuerst beschrieben, und dann eine .bersicht aller m.glichen Fehlermeldungen f.r jede Kategorie gegeben.

Compiler-Warnungen

Compiler-Warnungen zeigen ein potentielles Problem w.hrend einer Compilation an, das jedoch nicht schwerwiegend ist. Nach einer Warnung wird die Compilation fortgef.hrt, ohne da. die Generierung der Objekt-Datei dadurch betroffen w.re. F.r eine Warnung wird kein spezieller DOS Return-Code gesetzt.

Das generelle Format einer Compiler-Warnung lautet:

<Dateiname>(<Zeile>) Warning C1xxx <text>[: <Symbol>]

Compiler-Fehlermeldungen

Compiler-Fehlermeldungen zeigen Fehler an, die eine Compilation entscheidend beeinflussen k.nnen. Der Compiler wird versuchen, den Fehler zu behandeln, und fortzufahren. In den meisten F.llen wird jedoch eine Fortsetzung der Compilation nicht m.glich sein, und die Compilation wird ohne Generierung einer Objekt-Datei abgebrochen. Bei einem solchen Fehler wird ein DOS Return-Code von 1 zur.ckgegeben.
Das generelle Format einer Compiler-Fehlermeldung ist:

<Dateiname>(<Zeile>) Error C2xxx <text>[: <Symbol>]

Schwerwiegende Compiler-Fehlermeldungen

Schwerwiegende Compiler-Fehlermeldungen zeigen erhebliche Probleme w.hrend der Compilation einer Programm-Datei an. Daher wird die Compilation sofort beendet und keine Objekt-Datei erzeugt. Au.erdem wird 1 als DOS Return-Code zur.ckgegeben.
Das generelle Format einer schwerwiegenden Fehlermeldung ist:

<Dateiname>(<Zeile>) Fatal C3xxx <text>[: <Symbol>]

C1001 Return statement with no value in function

Problem: Innerhalb einer Funktionsdefinition wurde ein RETURN-Statement ohne R.ckgabewert angegeben. Unter Clipper sollte f.r jede Funktion ein R.ckgabewert definiert sein, auch wenn dieser bei einem Funktions-Aufruf nicht zugewiesen wird. Wird ein RETURN in einer Funktion ohne R.ckgabewert verwendet, liefert die Funktion NIL.

L.sung: .ndern Sie das Deklarations-Statement von FUNCTION in PROCEDURE oder f.gen Sie dem RETURN-Statement den fehlenden R.ckgabewert hinzu.

C1002 Procedure returns value

Problem: Innerhalb einer Prozedurdefinition wurde ein RETURN-Statement zusammen mit einem R.ckgabewert angegeben. Unter Clipper k.nnen nur Funktionen einen R.ckgabewert liefern, Prozeduren nicht.

L.sung: .ndern Sie die Deklaration von PROCEDURE in FUNCTION oder entfernen Sie den in Verbindung mit dem RETURN angegebenen R.ckgabewert aus dem Programm.

C1003 Ambiguous variable reference

Problem: Es wurde eine Referenz auf eine Variable durchgef.hrt, die zuvor nicht deklariert oder mit einem Alias versehen wurde.

L.sung: Die Variablen werden deklariert oder bei Datenfeldern mit FIELD oder einem ALIAS versehen. Dies soll auch als Unterst.tzung dienen, wenn nachtr.glich Variablen deklariert werden sollen.

C1004 Ambiguous variable reference, assuming memvar

Problem: Es wurde eine Referenz auf eine Variable durchgef.hrt, die zuvor nicht deklariert oder mit einem Alias versehen wurde. Das restliche Programm l..t den Compiler jedoch vermuten, da. es sich um eine MEMVAR-Deklaration handeln sollte.

L.sung: Deklarieren Sie die Variablen vor der Verwendung.

C1005 Redefinition or duplicate definition of #define

Problem: Dieselbe symbolische Konstante wurde in mehr als einer #define-Direktive definiert. Dies geschieht meist, wenn eine symbolische Konstante neu definiert wird, bevor eine alte Definition .ber #undef aufgehoben wurde.

L.sung: Obwohl es nicht unbedingt unzul.ssig ist, ein Symbol auf diese Weise umzudefinieren, so ist es doch die bessere Programmier-Technik, vorherige Definitionen zuvor .ber #undef aufzuheben.

C2001 Syntax Error: '<Anweisung>'

Problem: In der gemeldeten Anweisung ist ein Syntax-Fehler aufgetreten.

L.sung: Die korrekte Syntax f.r alle Befehle, Funktionen und Statements finden Sie in den beiden Referenz-Handb.chern.

C2002 Statement unterminated at end of line

Problem: Am Ende eines Statements ist ein Syntax-Fehler aufgetreten. Dies geschieht in der Regel, wenn ein Ausdruck oder eine Klammerung nicht vollst.ndig sind.

L.sung: Pr.fen Sie das Statement sorgf.ltig, korrigieren Sie es und compilieren Sie erneut.

C2003 Syntax error in statement: '<Statement>'

Problem: Irgendwo im angegebenen Statement ist ein Syntax-Fehler aufgetreten.

L.sung: Die korrekte Syntax f.r alle Befehle, Funktionen und Statements finden Sie in den beiden Referenz-Handb.chern.

C2004 Illegal character: '<Character>'

Problem: Au.erhalb einer Konstanten wurde ein unbekanntes Zeichen gefunden. Dies k.nnte zum Beispiel ein verirrtes Grafik-Zeichen innerhalb einer Anweisung sein.

L.sung: Entfernen Sie das Zeichen aus dem Programm und compilieren Sie erneut.

C2005 Statement not recognized, match failed at: '<code>'

Problem: Eine Anweisung konnte durch den Compiler nicht erkannt werden. Dies geschieht zum Beispiel, wenn ein Befehl vom Pre-Prozessor nicht erkannt wurde und daher ohne .bersetzung an den Compiler weitergegeben wurde.

L.sung: Korrigieren Sie die Anweisung und compilieren Sie erneut. Falls Sie eigene Befehle definiert haben, pr.fen Sie gegebenenfalls auch die zugeh.rigen Pre-Prozessor-Definitionen.

C2006 Statement not allowed outside procedure or function

Problem: Eine Programm-Datei wurde mit der Option /N compiliert, und eine andere Anweisung als FIELD, MEMVAR oder STATIC wurde vor der ersten Definition einer Prozedur oder Funktion gefunden.

L.sung: Entfernen Sie die falsch plazierte Anweisung und setzen Sie diese an eine entsprechende Position innerhalb einer Prozedur oder Funktion. Alternativ k.nnen Sie auch ohne diese Option compilieren, und Clipper damit veranlassen, eine Prozedur mit dem gleichen Namen wie das Programm zu erzeugen.

C2007 Unterminated string: '<String>'

Problem: Es wurde eine Zeichenfolge gefunden, bei welcher der abschlie.ende Begrenzer (" ']) fehlt. Dies kann durch die irrt.mliche Benutzung eines Apostrophs oder das versehentliche Weglassen eines Begrenzers bedingt sein.

L.sung: F.gen Sie den fehlenden Begrenzer hinzu. G.ltige Begrenzer sind einfache Anf.hrungszeichen ('), doppelte Anf.hrungszeichen (") und eckige Klammern ([]).

C2009 Invalid use of @ (pass by reference) operator: '<Parameter>'

Problem: Der pass by reference Operator wurde in Verbindung mit einer Variablen verwendet, die zuvor explizit mit dem FIELD-Statement deklariert wurde. Datenfelder m.ssen per Value und k.nnen auf keinen Fall per Referenz .bergeben werden.

L.sung: .bergeben Sie das Feld per Value. Da Felder allerdings eine globale G.ltigkeit besitzen, ist es eigentlich nicht notwendig, ein Feld explizit zu .bergeben.

C2010 Incorrect number of arguments

Problem: Eine der vordefinierten Clipper-Funktionen wurde mit einer ung.ltigen Anzahl von Argumenten aufgerufen. Eine Liste aller vordefinierten Clipper-Funktionen finden Sie im Anhang Reservierte Worte im Referenz-Handbuch II.

L.sung: Die korrekte Syntax aller Funktionen finden Sie im Kapitel Funktionen im Referenz-Handbuch II.

C2011 Exit statement with no loop in sight

Problem: Es wurde ein EXIT-Statement au.erhalb einer DO WHILE- oder FOR ... NEXT-Schleife gefunden. Das EXIT-Statement ist nur innerhalb dieser Schleifenkonstrukte g.ltig.

L.sung: Dieser Fehler resultiert oft aus grunds.tzlichen Problemen bei der Verschachtelung von Konstrukten. Durchsuchen Sie Ihr Programm also nach fehlenden DO WHILE- oder FOR ... NEXT-Statements oder entfernen Sie das betroffene EXIT.

C2012 LOOP statement with no loop in sight

Problem: Es wurde ein LOOP-Statement au.erhalb einer DO WHILE- oder FOR ... NEXT-Schleife gefunden. Das LOOP-Statement ist nur innerhalb dieser Schleifenkonstrukte g.ltig.

L.sung: Dieser Fehler resultiert oft aus grunds.tzlichen Problemen bei der Verschachtelung von Konstrukten. Durchsuchen Sie Ihr Programm also nach fehlenden DO WHILE- oder FOR ... NEXT-Statements oder entfernen Sie das betroffene LOOP.

C2013 EXIT statement violates enclosing sequence

Problem: Ein EXIT-Statement wurde zwischen dem BEGIN SEQUENCE und dem RECOVER innerhalb eines SEQUENCE-Anweisungsblockes gefunden, der sich wiederum innerhalb einer DO WHILE- oder FOR ... NEXT-Schleife befindet.

L.sung: Sie k.nnen eine SEQUENCE nicht derart verlassen. Stattdessen m.ssen Sie durch ein BREAK zum RECOVER oder END verzweigen und dort dann EXIT verwenden.

CDOW() Ermittelt den Wochentagsnamen eines Datums

Syntax

CDOW(<dDatum>) -> cWochentag

Parameter

<dDatum> ist der zu konvertierende Datumswert.

R.ckgabe

CDOW() gibt eine Zeichenkette zur.ck, die den Tag der Woche enth.lt. Der erste Buchstabe ist ein Gro.buchstabe, alle weiteren sind Kleinbuchstaben. Die maximale L.nge der Zeichenkette betr.gt neun Zeichen. Ein leerer Datumswert ergibt eine leere Zeichenkette ("").

Anwendung

CDOW() ist eine Datumskonvertierungsfunktion, die benutzt wird, um formatierte Ausgaben, wie Listen, Etiketten oder Bildschirmausgaben zu erzeugen.

Beispiele

Das Beispiel demonstriert den unterschiedlichen Gebrauch von CDOW():

```
? DATE()                    // Resultat: 13.07.90
? CDOW(DATE())              // Resultat: Freitag
? CDOW(DATE() + 7)          // Resultat: Freitag
? CDOW(CTOD("12.06.89"))    // Resultat: Montag
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

C2014 LOOP statement violates enclosing sequence

Problem: Ein LOOP-Statement wurde zwischen dem BEGIN SEQUENCE und dem RECOVER innerhalb eines SEQUENCE-Anweisungsblockes gefunden, der sich wiederum innerhalb einer DO WHILE- oder FOR ... NEXT-Schleife befindet.

L.sung: Sie k.nnen LOOP in dieser Form nicht einsetzen. Stattdessen m.ssen Sie durch ein BREAK zum RECOVER oder END verzweigen und dort dann LOOP verwenden.

C2015 Illegal initializer: '<Variable>'

Problem: Eine STATIC-Variable soll mit einem Wert initialisiert werden, der keine Konstante ist.

L.sung: Da STATIC-Variablen zur Compilier-Zeit aufgel.st und ihre Werte vor dem eigentlichen Programmstart zugewiesen werden, sind nur konstante Werte und einfache Operationen (keine Variablen oder Funktionen) m.glich. Andere Speicherklassen erlauben diese Art der Werte-Zuweisung.

C2016 Name conflicts with previous declaration: '<Variable>'

Problem: Eine PRIVATE-, PUBLIC- oder PARAMETERS-Variable soll den gleichen Namen wie eine .ber LOCAL-, STATIC-, FIELD- oder MEMVAR deklarierte Variable erhalten. Wenn Sie mit der Option /N compilieren, umfa.t das neben den Variablen mit eingeschr.nkter G.ltigkeit auch die mit Datei-weiter G.ltigkeit (filewide scope), die vor den ersten Funktions- oder Prozedur-Deklaration vorgenommen wurden.

L.sung: .ndern Sie den Variablennamen um.

C2017 Duplicate variable declaration: '<Variable>'

Problem: Eine Variable soll mit demselben Namen wie eine andere Variable dieses G.ltigkeitsbereiches deklariert worden sein. So lautet zum Beispiel eine STATIC-, gefolgt von einer LOCAL-Deklaration auf den gleichen Namen.

L.sung: Dies geschieht meist, wenn sehr viele Variablen in verschiedenen Statements deklariert werden, und ein Name versehentlich zweimal angegeben wird. Entfernen Sie die doppelte Deklaration.

C2018 Outer block variable out of reach

Problem: In einem verschachtelten Code-Block erh.lt ein innerer Block eine Referenz auf eine Variable, die in einem .u.eren Block deklariert wird.

L.sung: Setzen Sie die Deklaration in denselben Block, der auch die Variable referenziert.

C2019 CALL of Clipper procedure or function: '<Symbol>'

Problem: Es wird versucht, eine zuvor deklarierte Funktion oder Prozedur .ber CALL aufzurufen. CALL dient ausschlie.lich dazu, separat compilierte oder assemblierte Programme anderer Sprachen aufzurufen.

L.sung: Falls ein Namens-Konflikt mit einer externen Routine besteht, benennen Sie diese oder aber die Clipper-Routine um. Wenn beide Routinen nur in verschiedenen Modulen des Programmes angesprochen werden, k.nnen Sie auch die Clipper-Routine als STATIC deklarieren.

C2020 Mistreatment of CALLed symbol: '<Symbol>'

Problem: Eine zuvor .ber CALL aufgerufene Routine soll nun .ber DO oder einen Funktions-Aufruf angesprochen werden.

L.sung: Stellen Sie sicher, da. Routinen, die .ber CALL aufgerufen werden, nicht .ber die gleichen Namen wie Clipper-Routinen verf.gen.

C2021 Redefinition of Clipper procedure or function: '<Symbol>'

Problem: Es wurde eine PROCEDURE- oder FUNCTION-Deklaration f.r einen Namen gefunden, der bereits im selben G.ltigkeits-Bereich deklariert wurde.

L.sung: Prozeduren oder Funktionen unter Clipper sind entweder public (verf.gbar f.r alle anderen Module des Programmes) oder static (nur in dem Teil g.ltig, in dem die Deklaration erfolgte). Dieser Fehler tritt meist aufgrund von Namens-Konflikten zwischen zwei Public-Funktionen auf. Wenn beide Funktionen wirklich public sein m.ssen, so mu. eine der beiden umbenannt werden. Wenn jedoch beide als STATIC deklariert werden k.nnen, so ist dies unter Umst.nden vorzuziehen.

C2022 Redefinition of predefined function: '<Symbol>'

Problem: Es wurde eine Funktion mit dem selben Namen wie eine vordefinierte Clipper-Funktion angegeben. Eine komplette Liste dieser Funktionen finden Sie im Anhang Reservierte Worte im Referenz Handbuch II.

L.sung: Vordefinierte Funktionen von Clipper k.nnen nicht umdefiniert werden. Daher mu. die den Fehler ausl.sende Funktion umbenannt werden.

C2023 Clipper definition of CALLed symbol: '<Symbol>'

Problem: Der Name einer zuvor .ber CALL aufgerufenen Routine soll in einem PROCEDURE- oder FUNCTION-Statement deklariert werden.

L.sung: Dieser Konflikt kann nur aufgel.st werden, indem entweder die .ber CALL aufgerufene oder die zu deklarierende Routine umbenannt wird.

C2024 Unclosed control structures

Problem: Bei einer oder mehreren Kontroll-Strukturen fehlen END-Statements.

L.sung: .berpr.fen Sie alle BEGIN SEQUENCE, DO CASE, DO WHILE, FOR und IF-Statements, um sicher zu stellen, da. diese korrekt verschachtelt sind.

C2025 ELSE does not match IF

Problem: Ein ELSE-Statement wurde gefunden, das sich nicht innerhalb eines IF ... ENDIF-Konstruktes befindet.

L.sung: ELSE ist ein Teil der IF-Kontrollstruktur und kann daher nur innerhalb dieser vorkommen.

C2026 ELSEIF does not match IF

Problem: Ein ELSEIF-Statement wurde gefunden, das sich nicht innerhalb eines IF ... ENDIF-Konstruktes befindet.

L.sung: ELSEIF ist ein Teil der IF-Kontrollstruktur und kann daher nur innerhalb dieser vorkommen.

C2027 ENDIF does not match IF

Problem: Es wurde ein ENDIF ohne korrespondierendes IF gefunden.

L.sung: ENDIF ist ein Teil der IF-Kontrollstruktur und kann daher nur innerhalb dieser vorkommen. Dieser Fehler kann immer auch durch inkorrekt verschachtelte Strukturen bedingt sein.

C2028 ENDDO does not match WHILE

Problem: Es wurde ein ENDDO ohne ein korrespondierendes DO .. WHILE gefunden.

L.sung: ENDDO ist ein Teil der DO ... WHILE-Kontrollstruktur und kann daher nur innerhalb dieser vorkommen. Solch ein Fehler kann immer auch durch inkorrekt verschachtelte Strukturen bedingt sein.

C2029 NEXT does not match FOR

Problem: Es wurde ein NEXT ohne korrespondierendes FOR gefunden.

L.sung: NEXT ist ein Teil der FOR ... NEXT-Kontrollstruktur und kann daher nur als Teil einer solchen vorkommen. Dieser Fehler kann immer auch durch inkorrekt verschachtelte Strukturen bedingt sein.

C2030 ENDCASE does not match DO CASE

Problem: Es wurde ein ENDCASE ohne korrespondierendes DO CASE gefunden.

L.sung: CASE ist ein Teil der DO CASE ... ENDCASE-Kontrollstruktur und kann daher nur als Teil einer solchen vorkommen. Dieser Fehler kann immer auch durch inkorrekt verschachtelte Strukturen bedingt sein.

C2031 CASE or OTHERWISE is not immediately within DO CASE

Problem: Es wurde ein CASE- oder OTHERWISE-Statement gefunden, das sich au.erhalb eines DO CASE ... ENDCASE-Konstruktes befindet.

L.sung: CASE und OTHERWISE sind Teile der DO CASE ... ENDCASE-Kontrollstruktur und k.nnen daher nur als Teil einer solchen vorkommen.

C2032 TEXT statement error

Problem: In Verbindung mit einem TEXT-Statement ist ein Problem aufgetreten.

L.sung: Dieser Fehler k.nnte zum Beispiel auftreten, wenn Sie den TEXT-Befehl durch eine .bersetzungs-Direktive umdefinieren wollen und sich in dem resultierenden Code ein Fehler befindet.

C2033 Missing ENDTEXT

Problem: Es wurde ein TEXT-Statement ohne zugeh.riges ENDTEXT gefunden. Das TEXT-Konstrukt ist .hnlich der anderen Kontroll-Strukturen, es mu. durch ein ENDTEXT abgeschlossen werden.

L.sung: Dieser Fehler ist vermutlich aufgetreten, weil das ENDTEXT-Statement nicht angegeben wurde. F.gen Sie das Statement an der entsprechenden Stelle hinter dem Text-Block ein.

C2034 Formal parameters already declared

Problem: In einer Funktion oder Prozedur wird versucht, Parameter sowohl .ber das PARAMETERS-Statement als auch in Form von formalen Parametern (in Klammern) zu .bernehmen.

L.sung: Nur eine der beiden .bernahme-Methoden ist m.glich. Ordnen Sie die Varriablen einer Methode der jeweils anderen zu.

C2035 Invalid declaration

Problem: In einer Variablen-Deklaration (LOCAL, MEMVAR, etc.) wurde ein Fehler entdeckt.

L.sung: .berpr.fen Sie die jeweilige Deklaration auf ihre Syntax.

C2036 Mayhem in CASE handler

Problem: Zwischen dem DO CASE und dem ersten CASE oder OTHERWISE wurden andere Statements gefunden. Hier sind keine anderen Statements erlaubt.

L.sung: Entfernen Sie die Statements oder plazieren Sie diese in einem der CASE-Bl.cke.

C2037 Invalid procedure name in DO statement

Problem: In einem DO-Statement wurde ein Argument entdeckt, welches kein g.ltiger Prozedur-Name ist.

L.sung: Unter Clipper m.ssen alle Bezeichner mit einem Buchstaben oder einem Unterstrich beginnen und aus Buchstaben, Ziffern oder Unterstrichen bestehen.

C2038 Invalid target name in CALL statement

Problem: In einem CALL-Statement wurde ein Argument entdeckt, welches kein g.ltiger Prozedur-Name ist.

L.sung: Unter Clipper m.ssen alle Bezeichner mit einem Buchstaben oder einem Unterstrich beginnen und aus Buchstaben, Ziffern oder Unterstrichen bestehen.

C2039 Invalid selector in send

Problem: An eine Objekt-Klasse wurde .ber den :-Operator eine unbekannte message geschickt. Meist handelt es sich in solchen F.llen um eine fehlerhafte Schreibweise der message, oder diese wurde an ein falsches Objekt geschickt.

L.sung: Im Kapitel Objekt-Klassen des Referenz-Handbuches II finden Sie eine komplette .bersicht aller Objekt-Klassen.

C2040 Invalid unary inline operator

Problem: Der betroffene Ausdruck wurde nicht korrekt formuliert.

L.sung: Pr.fen Sie den Ausdruck sorgf.ltig und korrigieren Sie die Fehler. Benutzen Sie gegebenenfalls zus.tzliche Klammern, um den Ausdruck .bersichtlicher zu gestalten - dies vereinfacht die Fehlersuche unter Umst.nden erheblich.

C2041 Invalid binary operator

Problem: Der betroffene Ausdruck wurde nicht korrekt formuliert.

L.sung: Pr.fen Sie den Ausdruck sorgf.ltig und korrigieren Sie die Fehler. Benutzen Sie gegebenenfalls zus.tzliche Klammern, um den Ausdruck .bersichtlicher zu gestalten - dies vereinfacht die Fehlersuche unter Umst.nden erheblich.

C2042 Invalid lvalue

Problem: Der Teil auf der linken Seite eines Zuweisungs-Operators ist ein ung.ltiger Bezeichner. Nur g.ltigen Bezeichnern k.nnen Werte oder Referenzen zugewiesen werden.

L.sung: Dieser Fehler kann zum Beispiel auftreten, wenn in einem Vergleich anstelle des == Operators der := Operator verwendet wurde (5 := y + 10).

C2043 Invalid alias expression

Problem: Der Bezeichner auf der linken Seite eines Alias-Operators ist kein g.ltiger Bezeichner. Auf der linken Seite eines Alias-Operators kann sich nur ein g.ltiger Bezeichner oder ein Makro-Ausdruck befinden.

L.sung: Pr.fen Sie, ob Sie keinen Ausdruck als Alias-Namen angegeben haben.

C2044 Invalid function name

Problem: Es wurde ein Funktions-Aufruf entdeckt, bei dem kein g.ltiger Funktionsname verwendet wurde.

L.sung: Unter Clipper m.ssen alle Bezeichner mit einem Buchstaben oder einem Unterstrich beginnen und aus Buchstaben, Ziffern oder Unterstrichen bestehen.

C2045 Target name was used previously in non-CALL context

Problem: Es wurde ein CALL-Aufruf einer Prozedur oder Funktion gefunden, die bereits zuvor .ber DO oder als Funktion aufgerufen worden ist.

L.sung: Stellen Sie sicher, da. die Routinen, die .ber CALL aufgerufen werden, nicht .ber die gleichen Namen wie Clipper-Routinen verf.gen.

C2046 SEQUENCE nesting error

Problem: Eine BEGIN SEQUENCE . END-Kontrollstruktur wurde fehlerhaft verschachtelt.

L.sung: Kontrollstrukturen m.ssen entsprechend den Definitionen immer korrekt verschachtelt werden. Hier k.nnen au.erdem BREAK und RECOVER nicht au.erhalb einer BEGIN SEQUENCE . END-Struktur verwendet werden. Pr.fen Sie, ob das BEGIN SEQUENCE-Konstrukt .ber ein zugeh.riges END verf.gt und auch alle darin verschachtelten Konstrukte korrekt abgeschlossen wurden.

C2047 GET contains complex macro

Problem: Es wurde der Versuch erkannt, ein GET auf einen Makro-Ausdruck anzuwenden (z.B. &(<Ausdruck>)).

L.sung: Weisen Sie den Makro-Ausdruck einer Variablen zu und f.hren Sie die Eingabe dann auf diese Variable durch.

C2048 GET contains both macro and declared symbol references

Problem: Es wurde versucht, auf eine deklarierte GET-Variable mit dem Makro Operator zuzugreifen.

L.sung: Weisen Sie den Ausdruck einer Variablen zu, auf die Sie dann das GET ausf.hren.

C2049 Code block contains complex macro

Problem: Es wurde versucht, einen Makroausdruck (z.B. &(<Ausdruck>)) in einem Code-Block einzusetzen.

L.sung: Weisen Sie den Makroausdruck einer Variablen zu und benutzen Sie diese im Code-Block.

CHR() Wandelt ASCII-Code in das korrespondierende Zeichen um

Syntax

CHR(<nAscii>) -> cZeichen

Parameter

<nAscii> ist der ASCII-Code im Bereich von 0 bis 255.

R.ckgabe

CHR() gibt ein einzelnes Zeichen zur.ck, das durch den Code in <nAscii> definiert worden ist.

Anwendung

CHR() ist eine numerische Konvertierungsfunktion, welche einen ASCII-Code in das korrespondierende Zeichen umwandelt. Sie arbeitet umgekehrt wie die Funktion ASC().

CHR() ist eine sehr vielseitige Funktion, die eine Reihe von Aufgaben erf.llt:

. Steuersequenzen an den Drucker senden
. Einen Ton erzeugen
. Darstellen grafischer Sonderzeichen auf Drucker oder Bildschirm
. INKEY()-Werte in ein Zeichen umwandeln
. Zeichen im Tastaturpuffer ablegen

Hinweise

. Ton erzeugen: Befehle f.r die Konsole (z.B. ? oder ??) behandeln ASCII-Kontrollzeichen wie BELL, CR und LF anders als andere Ausgabebefehle (z.B. @...SAY), bei denen Steueranweisungen der Kontrollzeichen ignoriert werden. Die Anweisung @ ...SAY CHR(7) zeigt ein Grafikzeichen auf den Bildschirm, w.hrend ?? CHR(7) einen Ton erzeugt.

. Null Zeichen: In Zeichenfolgen kennzeichnet das Zeichen CHR(0) das Ende dieser Folge. Das einzelne Zeichen CHR(0) hingegen hat die L.nge 1 und wird wie jedes andere Zeichen behandelt. Das erm.glicht, dieses Zeichen an beliebige Ger.te und Dateien zu senden.

Beispiele

. Das Beispiel zeigt den Einsatz von CHR() mit verschiedenen Argumenten:

```
? CHR(72)                    // Resultat: H
? CHR( ASC( 61) + 32)        // Resultat: =
? CHR(7)                     // Resultat: ein Ton
```

. Das folgende Beispiel zeigt Ihnen den Unterschied zwischen dem Null-Zeichen und einer leeren Zeichenkette:

```
? LEN("")                    // Resultat: 0
? LEN(CHR(0))                // Resultat: 1
```

. Im n.chsten Beispiel gibt die Funktion INKEY() den ASCII-Wert einer gedr.ckten Taste zur.ck. Es wird CHR() benutzt, um die Bedingungen leichter lesen zu k.nnen.

```
@ 23, 0 SAY "Um die Nachricht zu l.schen, Taste S bet.tigen.."
i := 32
DO WHILE .NOT. CHR(i) $ "Ss"
  i := INKEY(0)
ENDDO
@ 23, 0
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

C2050 Code block contains both macro and declared symbol references

---

Problem: In einem Code-Block wird der Makrooperator auf eine deklarierte Variable angewandt.

L.sung: Weisen Sie den Makroausdruck einer Variablen zu, und benutzen Sie diese im Code-Block.

C2051 LOCAL declaration follows executable statement

Problem: Ein LOCAL-Statement tritt nach einem ausf.hrbaren Statement auf. Variablen-Deklarationen m.ssen vor jedem ausf.hrbaren Programmausdruck erfolgen.

L.sung: Versetzen Sie das LOCAL-Statement an den Beginn der Prozedur- oder Funktionsdefinition vor alle anderen Anweisungen.

C2052 MEMVAR declaration follows executable statement

Problem: Ein MEMVAR-Statement tritt nach einem ausf.hrbaren Statement auf. Variablen-Deklarationen m.ssen vor jedem ausf.hrbaren Programmausdruck geschehen.

L.sung: Setzen Sie das MEMVAR-Statement an den Beginn der Prozedur- oder Funktionsdefinition vor alle anderen Anweisungen.

C2053 FIELD declaration follows executable statement

Problem: Ein FIELD-Statement tritt nach einem ausf.hrbaren Statement auf. Variablen-Deklarationen m.ssen vor jedem ausf.hrbaren Programmausdruck geschehen.

L.sung: Versetzen Sie das FIELD-Statement an den Beginn der Prozedur- oder Funktionsdefinition vor alle anderen Anweisungen.

C2054 STATIC declaration follows executable statement

Problem: Ein STATIC-Statement tritt nach einem ausf.hrbaren Statement auf. Variablen-Deklarationen m.ssen vor jedem ausf.hrbaren Programmausdruck geschehen.

L.sung: Versetzen Sie das STATIC-Statement an den Beginn der Prozedur- oder Funktionsdefinition vor alle anderen Anweisungen.

C2055 Syntax error in #define

Problem: Eine #define-Direktive enth.lt einen Syntaxfehler

L.sung: Die Syntax f.r diese und alle anderen Pre-Prozessor-Direktiven
wird im Kapitel Direktiven im Referenz-Handbuch I beschrieben.

C2056 Unexpected end of file in #define

Problem: Eine #define-Direktive wird durch das Dateiende vorzeitig beendet.

L.sung: Die Syntax f.r diese und alle anderen Pre-Prozessor-Direktiven wird im Kapitel Direktiven im Referenz-Handbuch I beschrieben.

C2057 Label missing in #define

Problem: Eine #define-Direktive enth.lt keinen Bezeichner.

L.sung: Die Syntax f.r diese und alle anderen Pre-Prozessor-Direktiven
wird im Kapitel Direktiven im Referenz-Handbuch I beschrieben.

C2058 Comma or right parenthesis missing in #define

Problem: Eine Pseudofunktion mit der #define-Direktive enth.lt entweder kein Komma oder keine abschlie.ende Klammer.

L.sung: Eine Pseudofunktions-Definition mu. eine entsprechende Anzahl von offenen und geschlossenen Klammern bzw. Kommata zur Argument-Trennung besitzen. Korrigieren Sie die entsprechende Definition.

C2059 Missing => in #translate/#command

Problem: Eine #translate- oder #command-Direktive enth.lt keinen => Separator.

L.sung: Jede #translate-/#command-Direktive mu. das Symbol => enthalten, auch wenn die Definition keine Zuweisung beinhaltet. Die Syntax f.r diese und alle anderen Pre-Prozessor-Direktiven wird im Kapitel Direktiven im Referenz-Handbuch I beschrieben.

C2060 Unknown result marker in #translate/#command

Problem: Eine #translate- oder #command-Direktive enth.lt einen Verweis auf einen result-marker im result-pattern, der im match-pattern der Direktive nicht definiert wurde.

L.sung: .berpr.fen Sie die Schreibweise des result-markers, um sicherzustellen, da. ein entsprechender match-marker gefunden wird.

C2061 Label error in #translate/#command

Problem: Eine #translate- oder #command-Direktive enth.lt einen Verweis auf einen fehlerhaften match-marker.

L.sung: Die Syntax f.r diese und alle anderen Pre-Prozessor-Direktiven wird im Kapitel Direktiven im Referenz-Handbuch I beschrieben.

C2062 Bad match marker in #translate/#command

Problem: Eine #translate- oder #command-Direktive enth.lt einen Verweis auf einen unzul.ssigen match-marker.

L.sung: Die Syntax f.r diese und alle anderen Pre-Prozessor-Direktiven wird im Kapitel Direktiven im Referenz-Handbuch I beschrieben.

C2063 Bad result marker #translate/#command

Problem: Eine #translate- oder #command-Direktive enth.lt einen Verweis auf einen unzul.ssigen result-marker.

L.sung: Die Syntax f.r diese und alle anderen Pre-Prozessor-Direktiven wird im Kapitel Direktiven im Referenz-Handbuch I beschrieben.

C2064 Bad restricted match marker in #translate/#command

Problem: Eine #translate- oder #command-Direktive enth.lt einen Verweis auf einen unzul.ssig begrenzten match-marker.

L.sung: Die Syntax f.r diese und alle anderen Pre-Prozessor-Direktiven wird im Kapitel Direktiven im Referenz-Handbuch I beschrieben.

C2065 Empty optional clause in #translate/#command

Problem: Eine #translate- oder #command-Direktive enth.lt eine leere optionale Klausel.

L.sung: Optionale Klauseln werden in #command- und #translate-Direktiven durch eckige Klammern ([ ]) gekennzeichnet. Diese Fehlermeldung weist auf ein solches Klammernpaar ohne Inhalt, also ohne Klausel, hin.

C2066 Unclosed optional clause in #translate/#command

Problem: Eine #translate- oder #command-Direktive enth.lt eine optionale Klausel, bei der die abschlie.ende eckige Klammer fehlt.

L.sung: Alle optionalen Klauseln m.ssen korrekt in ein Paar eckiger Klammern eingeschlossen sein.

C2067 Too many nested #ifdefs

Problem: Die maximale Anzahl von 16 Verschachtelungsebenen f.r #ifdef-Direktiven wurde .berschritten.

L.sung: Die Verschachtelungstiefe mu. reduziert werden.

C2068 Error in #ifdef

Problem: Eine #ifdef-Direktive enth.lt einen Fehler.

L.sung: Die Syntax f.r diese und alle anderen Pre-Prozessor-Direktiven
wird im Kapitel Direktiven im Referenz-Handbuch I beschrieben.

C2069 #endif does not match #ifdef

Problem: F.r eine #endif-Direktive wurde keine passende #ifdef- oder #ifndef-Anweisung gefunden.

L.sung: Pr.fen Sie die Struktur der einzelnen #ifdef- oder #ifndef-Verschachtelungen.

C2070 #else does not match #ifdef

Problem: F.r eine #else-Direktive wurde keine passende #ifdef- oder #ifndef-Anweisung gefunden.

L.sung: Eine #else-Direktive mu. immer in eine #ifdef/#ifndef - #endif-Anweisung verschachtelt sein.

C2071 Error in #undef

Problem: Eine #undef-Direktive enth.lt einen Fehler.

L.sung: Die Syntax f.r diese und alle anderen Pre-Prozessor-Direktiven
wird im Kapitel Direktiven im Referenz-Handbuch I beschrieben.

C2072 Ambiguous match pattern in #translate/#command

Problem: Eine #translate- oder #command-Direktive enth.lt zwei aufeinanderfolgende optionale Klauseln ohne verbindendes Schl.sselwort.

L.sung: Dies ist nicht erlaubt, da der Pre-Prozessor keine M.glichkeit hat, den korrekten match-marker zu ermitteln. Beispiele f.r die #translate- oder #command-Direktive finden Sie im Kapitel Direktiven im Referenz-Handbuch I.

C2073 Result pattern contains nested clauses in #translate/#command

---

Problem: Eine #translate- oder #command-Direktive enth.lt verschachtelte result-clauses, z.B. verschachtelte eckige Klammern.

L.sung: Geschachtelte Klauseln sind in result-pattern nicht erlaubt. In einem solchen Fall sollten Sie zwei separate Regeln einsetzen.

C2075 Too many locals

Problem: In einer Funktions- oder Prozedur-Definition wurde die maximale Anzahl von 255 LOCAL-Variablen .berschritten.

L.sung: Der Einsatz so vieler LOCAL-Variablen deutet darauf hin, da. die Routine zu komplex ist. Teilen Sie diese in mehrere Routinen auf und reduzieren Sie so die Anzahl der LOCAL-Deklarationen pro Routine.

C2076 Too many parameters

Problem: In einer Funktions- oder Prozedur-Definition wurde die maximale Anzahl von 255 deklarierten Parametern .berschritten.

L.sung: Der Einsatz so vieler Parameter deutet darauf hin, da. die Routine zu komplex ist. Teilen Sie diese in mehrere Routinen auf und reduzieren Sie so die Anzahl der deklarierten Parameter pro Routine.

C2077 Too many parameters

Problem: In einem PARAMETER-Statement wurde die maximale Anzahl von 255 Parametern pro Funktion/Prozedur .berschritten.

L.sung: Der Einsatz so vieler Parameter deutet daraufhin, da. die Routine zu komplex ist. Teilen Sie diese in mehrere Routinen auf und reduzieren Sie so die Anzahl der Parameter pro Routine.

C2078 Circular #define

Problem: Eine #define-Direktive enth.lt rekursive Definitionen, z.B. #define x y gefolgt von einem #define y x.

L.sung: Eine rekursive Definition mit #define kann vom Pre-Prozessor nicht aufgel.st werden und ist daher nicht erlaubt.

C2079 Circular #translate/#command

Problem: Eine #translate- oder #command-Direktive enth.lt rekursive Definitionen.

L.sung: Eine rekursive Definition mit #translate oder #command kann vom Pre-Prozessor nicht aufgel.st werden und ist daher nicht erlaubt. Beachten Sie, da. beide Direktiven immer 4 Buchstaben lange Abk.rzungen der Schl.sselw.rter erkennen, wodurch eine nicht erw.nschte rekursive Definition und dadurch dieser Fehler auftauchen kann.

C3001 Out of memory

Problem: Der Compiler kann nicht auf gen.gend Speicher zugreifen.

L.sung: Die einfachste M.glichkeit besteht darin, dem Compiler, zum Beispiel durch das Entfernen von TSR-Programmen, mehr Speicher zur Verf.gung zu stellen. Desweiteren sollten Sie versuchen, die Komplexii.t des Programmes zu verringern, indem die Anzahl der Prozeduren und Funktionen im Quelltext (.PRG) oder die Anzahl der Quelltexte in der Script-Datei reduziert wird. Auch die Verringerung der Pre-Prozessor-Direktiven kann das Problem beseitigen.

C3002 Input buffer overflow

Problem: Es soll ein Statement verarbeitet werden, das f.r den Pre-Prozessor zu lang ist.

L.sung: Teilen Sie das Statement in mehrere kleinere Statements auf.

C3003 Can't open intermediate file

Problem: Der Compiler erzeugte eine tempor.re Datei, um Zwischenresultate abzuspeichern, und kann diese Datei jetzt nicht mehr .ffnen.

L.sung: Dieser Fehler ist sehr unwahrscheinlich. Er kann eigentlich nur dann auftreten, wenn sich die Zugriffsrechte auf diese Datei ge.ndert haben (im Netzwerk m.glich) oder physikalische Laufwerksprobleme aufgetreten sind.

C3004 Bad command line option: '<Option>'

Problem: In der Compiler-Kommandozeile wurde eine ung.ltige oder unbekannte Option angegeben.

L.sung: F.hren Sie CLIPPER.EXE ohne Argumente aus. Sie erhalten eine Liste aller verf.gbaren Optionen.

C3005 Bad command line parameter: '<Option>'

Problem: In der Compiler-Kommandozeile wurde ein ung.ltiger oder nicht korrekter Parameter angegeben.

L.sung: F.hren Sie CLIPPER.EXE ohne Argumente aus, um die Syntaxbeschreibung der Kommandozeile zu erhalten.

C3006 Can't create preprocessed output file

Problem: Die /P-Option wurde angegeben, aber der Compiler kann keine Ausgabedatei erzeugen.

L.sung: Diese Fehlermeldung kann mehrere Gr.nde haben: keine freien Datei-Handles, nicht gen.gend freier Plattenplatz oder unzureichende Zugriffsrechte im Netzwerk.

C3007 Can't open #include file

Problem: Der Pre-Prozessor kann die mit #include angegebene Datei nicht .ffnen. Der Dateiname ist vielleicht falsch geschrieben worden oder die Datei ist in keinem der, durch die Environment-Variable INCLUDE bestimmten Verzeichnisse zu finden.

L.sung: Korrigieren Sie den Dateinamen oder stellen Sie sicher, da. die Datei gefunden werden kann (.ndern der Environment-Variablen bzw. der /I Option des Compilers).

C3008 Bad filename in #include

Problem: In einer #include-Direktive wurde ein Dateiname nicht als literale Zeichenkette in Anf.hrungszeichen angegeben (z.B. "INIT.CH"). Spitze Klammern (<>) werden nicht unterst.tzt.

L.sung: Korrigieren Sie die Schreibweise des Dateinamens.

C3009 Too many nested #includes

Problem: Die maximale Anzahl von verschachtelten #include-Direktiven wurde .berschritten.

L.sung: Clipper 5.0 unterst.tzt 16 Verschachtelungsebenen bei #include.

C3010 Invalid name follows #

Problem: Eine unbekannte bzw. ung.ltige Direktive wurde angegeben.

L.sung: Entweder wurde die Direktive falsch geschrieben, oder diese Direktive wird nicht vom Clipper Pre-Prozessor unterst.tzt. Eine komplette Liste aller Direktiven wird im Kapitel Direktiven im Referenz-Handbuch I beschrieben.

C3011 Can't open standard rule file

Problem: Der Compiler kann die Standard Header-Datei (im Allgemeinen STD.CH, bzw. die mit /U spezifizierte Datei) nicht .ffnen.

L.sung: Der Compiler sucht diese Datei, wie jede andere Header-Datei, zuerst im aktuellen Verzeichnis und dann in dem mit INCLUDE bestimmten Pfad. .ndern Sie die Environment-Variable INCLUDE bzw. geben Sie mit der Option /I eine korrigierte Pfadliste an.

C3012 Too many standard rules

Problem: In der Standard Header-Datei (im Allgemeinen STD.CH, bzw. die mit /U spezifizierte Datei) existieren zu viele Regeln.

L.sung: Verkleinern Sie die Standard Header-Datei und erstellen Sie eine zweite Header-Datei, welche die anderen Regeln enth.lt. Diese Header-Datei mu. dann in jede Quelldatei der Applikation mit #include eingebunden werden.

C3013 Expression stack overflow

Problem: Dies tritt normalerweise nur bei extrem gro.en Prozeduren bzw. Funktionen oder bei sehr komplexen Ausdr.cken auf.

L.sung: Teilen Sie die Prozedur, Funktion oder den Ausdruck auf.

C3014 Expression stack underflow

Problem: Dies tritt normalerweise nur bei extrem gro.en Prozeduren bzw. Funktionen oder bei sehr komplexen Ausdr.cken auf.

L.sung: Teilen Sie die Prozedur, Funktion oder den Ausdruck auf.

C3015 Control stack overflow

Problem: Dieser Fehler kann durch zu viele verschachtelte Kontrollstrukturen hervorgerufen werden.

L.sung: Untersuchen Sie das Programm und versuchen Sie, die Kontrollstruktur zu vereinfachen.

C3016 Control stack underflow

Problem: Dieser Fehler kann durch zu viele verschachtelte Kontrollstrukturen hervorgerufen werden.

L.sung: Untersuchen Sie das Programm und versuchen Sie, die Kontrollstruktur zu vereinfachen.

C3017 Error reading or opening script file

Problem: Der Compiler kann die Script-Datei nicht finden oder nicht .ffnen.

L.sung: Der Compiler sucht die Script-Datei nur im aktuellen Verzeichnis. Stellen Sie sicher, da. die Datei dort existiert und da. Sie die Dateierweiterung .CLP besitzt. Andernfalls m.ssen Sie die Datei mit kompletter Namens- und Pfadangabe spezifizieren.

C3018 Too many symbols

Problem: Die Programm-Datei(en) in der aktuellen Objekt-Datei enthalten zu viele Symbole.

L.sung: Vereinfachen Sie das Programm, so da. es weniger Symbole ben.tigt (z.B. Wiederverwenden von Variablennamen anstatt st.ndig neue zu definieren).

C3019 Too many publics

Problem: Dieser Fehler kann dann auftreten, wenn zu viele Programm-Dateien auf einmal compiliert werden sollen.

L.sung: Teilen Sie die Programm-Dateien auf verschiedene Compiler-L.ufe auf.

C3020 Too many segments

Problem: Dieser Fehler kann dann auftreten, wenn zu viele Programm-Dateien auf einmal compiliert werden

L.sung: Teilen Sie die Programm-Dateien auf verschiedene Compiler-L.ufe auf.

C3021 Too many fixups

Problem: Dieser Fehler kann dann auftreten, wenn zu viele Programm-Dateien auf einmal compiliert werden

L.sung: Teilen Sie die Programm-Dateien auf verschiedene Compiler-L.ufe auf.

C3022 Too many external references

Problem: Dieser Fehler kann dann auftreten, wenn zu viele Programm-Dateien auf einmal compiliert werden sollen.

L.sung: Teilen Sie die Programm-Dateien auf verschiedene Compiler-L.ufe auf.

C3023 Too many labels

Problem: Dieser Fehler kann dann auftreten, wenn zu viele Programm-Dateien auf einmal compiliert werden

L.sung: Teilen Sie die Programm-Dateien auf verschiedene Compiler-L.ufe auf.

C3024 Too many procs

Problem: Dieser Fehler kann dann auftreten, wenn zu viele Programm-Dateien auf einmal compiliert werden

L.sung: Teilen Sie die Programm-Dateien auf verschiedene Compiler-L.ufe auf.

C3025 Too many proc request

Problem: Dieser Fehler kann dann auftreten, wenn zu viele Programm-Dateien auf einmal compiliert werden sollen.

L.sung: Teilen Sie die Programm-Dateien auf verschiedene Compiler-L.ufe auf.

C3026 Segment too big

Problem: Dieser Fehler kann dann auftreten, wenn zu viele Programm-Dateien auf einmal compiliert werden

L.sung: Teilen Sie die Programm-Dateien auf verschiedene Compiler-L.ufe auf.

C3027 Proc too big

Problem: Dieser Fehler kann dann auftreten, wenn zu viele Programm-Dateien auf einmal compiliert werden

L.sung: Teilen Sie die Programm-Dateien auf verschiedene Compiler-L.ufe auf.

C3028 Symbol table too big

Problem: Dieser Fehler kann dann auftreten, wenn zu viele Programm-Dateien auf einmal compiliert werden sollen.

L.sung: Teilen Sie die Programm-Dateien auf verschiedene Compiler-L.ufe auf.

C3029 Write error to intermediate file

Problem: Der Compiler hat eine tempor.re Datei erzeugt, um Zwischenresultate abzulegen, und kann nicht in diese Datei schreiben.

L.sung: Dieser Fehler kann auftreten, wenn w.hrend des Compiler-Laufes die Plattenkapazit.t ersch.pft ist. Mit Hilfe der Option /T kann die tempor.re Datei auf einem anderen Laufwerk erzeugt werden.

C3030 Write error to OBJ

Problem: Der Compiler hat eine Objekt-Datei erzeugt und kann nicht in diese Datei schreiben.

L.sung: Dieser Fehler kann auftreten, wenn w.hrend des Compiler-Laufes die Plattenkapazit.t ersch.pft ist. Mit Hilfe der /O-Option k.nnen die Objekt-Dateien auf einem anderem Laufwerk erzeugt werden.

C3031 Can't create OBJ

Problem: Der Compiler kann eine Objekt-Datei nicht erzeugen.

L.sung: Dieser Fehler kann auftreten, wenn z.B. keine Verzeichniseintr.ge mehr frei oder im Netzwerk nicht die entsprechenden Rechte vorhanden sind. Mit Hilfe der /O-Option k.nnen die Objekt-Dateien auf einem anderem Laufwerk oder Verzeichnis erzeugt werden.

C3032 Can't create intermediate file

Problem: Der Compiler versucht, eine tempor.re Datei zu erzeugen, um Zwischenergebnisse abzuspeichern. Dieser Versuch schl.gt fehl.

L.sung: L.schen Sie nicht ben.tigte Dateien um Platz f.r die tempor.re Datei zu schaffen.

.RTLink-Warnungen

.RTLink-Fehlermeldungen

Sonstige Fehlermeldungen von .RTLink

.RTLink-Fehlermeldungen

In diesem Abschnitt werden die m.glichen Fehlermeldungen, die w.hrend eines Linklaufs mit RTLink entstehen k.nnen, zusammengefa.t. Die Meldungen werden nach ihrer Wichtigkeit und dem Zeitpunkt im Linkvorgang in verschiedene Kategorien unterteilt. Jede dieser Kategorien wird zusammen mit den entsprechenden Fehlermeldungen weiter unten beschrieben. Informationen .ber die Syntax und Operationen des Linkers finden Sie im Kapitel Der Clipper Linker - RTLINK.EXE.
Mit den hier vorgestellten Fehlermeldungen und Warnungen werden die h.ufigsten Probleme, die w.hrend des Linkvorganges auftauchen k.nnen, abgedeckt.

.RTLink-Warnungen

In der ersten Kategorie sind die von .RTLink erzeugten Warnungen zusammengefa.t. Diese Warnungen weisen auf potentielle Fehler w.hrend der Programmausf.hrung hin, brechen den Linkvorgang allerdings nicht ab. Der DOS-R.ckgabewert wird nur dann auf 1 gesetzt, wenn die Warnung wrt0022 (unresolved symbol) erscheint, ansonsten beh.lt der DOS-R.ckgabewert seinen normalen Wert bei.
Die allgemeine Form einer .RTLink-Warnung ist:

```
warning wrtxxxx: Meldung
```

.RTLink-Fehlermeldungen

.RTLink-Fehlermeldungen bezeichnen Probleme, die den Linkvorgang zum Abbruch bringen, so da. keine ausf.hrbare .EXE-Datei erzeugt wird. Ursachen k.nnen dabei z.B. sein: eine ben.tigte Datei wird nicht gefunden, ein Syntaxfehler in der .RTLink-Kommandozeile oder ein Fehler in einer .OBJ- oder .LIB-Datei wurde gefunden. In diesem Fall wird der DOS-R.ckgabewert auf 1 gesetzt.
Die allgemeine Form einer .RTLink-Fehlermeldung ist:

```
error ertxxxx: Meldung
```

Sonstige Fehlermeldungen von .RTLink

Diese Kategorie beinhaltet spezielle Fehlermeldungen von .RTLink. Nach einem solchen Fehler kann der Linkvorgang normalerweise nicht beendet werden, so da. keine ausf.hrbare .EXE-Datei erzeugt wird. In diesem Fall wird der DOS-R.ckgabewert auf 1 gesetzt.
Die allgemeine Form einer .RTLink-Fehlermeldung dieser Kategorie ist:

```
error eutxxxx: Meldung
```

wrt0003 memory model mismatch im COMENT record
('<vorheriges Modell>'!= <angetroffenes Modell>,
COMENT record at 0x<offset>, file '<Datei>'

Problem: Es gibt verschiedene Typen von COMENT-S.tzen, die eine beschreibende Information .ber eine Objekt-Datei beinhalten, aber im allgemeinen die Ausgabe des Linkers nicht beeinflussen. Diese S.tze sind mit den Kommentarzeilen in einem Programmquelltext zu vergleichen. Es gibt z.B. einen COMENT-Typ, der das Speichermodell bestimmt. Dadurch wird allerdings der Linklauf von .RTLink nicht beeinflu.t.
Diese Meldung weist darauf hin, das der Datenteil des COMENT-Satzes nicht zu den vom Linker vorher angetroffenen Daten pa.t. Da das Format dieses COMENT-Satzes nicht ver.ffentlicht ist und viele Compilerhersteller zus.tzliche Informationen in diese COMENT-S.tze einbringen, deutet die Meldung normalerweise nicht auf einen Fehler hin. Sie weist nur auf unterschiedliche Daten in den beiden Datens.tzen der Objekt-Datei hin.

L.sung: Es mu. sicher gestellt werden, da. alle Objekt-Dateien und Bibliotheken im selben Speichermodell erstellt worden sind. Ist das der Fall, so kann diese Warnung ignoriert werden.

wrt0004 Duplicate start adress (using the first one we saw on this level)

Problem: Einige Programmiersprachen definieren in jeder erzeugten Objekt-Datei eine Startadresse f.r das Programm. Die im ersten Objekt gefundene Startadresse wird von .RTLink als Startadresse f.r das Programm benutzt. Diese Warnung teilt mit, da. mehrere Definitionen der Startadresse gefunden wurden und die im aktuellen Modul gefundene Startadresse ignoriert wird.

L.sung: Ist in einer angesprochenen .PLL-Datei eine Startadresse definiert und der Linker findet eine neue Startadresse in einer .OBJ-Datei, so wird die Startadresse in der .PLL-Datei ignoriert. Durch das Kommando VERBOSE kann festgestellt werden, welche Datei gerade bearbeitet wird, wenn die Warnung ausgegeben wird.

wrt0006 No Stack Segment

Problem: F.r das erzeugte Programm wurde kein Stacksegment definiert. Ob diese Warnung ignoriert werden kann oder nicht, h.ngt vom Programm selbst ab. Normalerweise haben nur Assemblerprogramme kein Stacksegment. In h.heren Programmiersprachen weist diese Meldung auf einen schwereren Fehler hin.

L.sung: Soll das Programm nach dem Linkvorgang noch mit EXE2BIN bearbeitet werden, kann diese Meldung normalerweise ignoriert werden. Andernfalls mu. ein Stacksegment definiert werden.

wrt0009 Unrecognized map type '<char>'....ignored

Problem: Eine der angegebenen Spezifikationen in der MAP-Option ist nicht zul.ssig.

L.sung: Die einzigen g.ltigen Spezifikationen f.r die MAP-Option sind A, N und S, die in Gro.- oder Kleinbuchstaben angegeben werden k.nnen. Informationen .ber die MAP-Option finden Sie im Kapitel Der Clipper Linker - RTLINK.EXE.

wrt0011 Public symbol '<Symbol>' redefined

Problem: Diese Meldung ist besonders im Zusammenhang mit .PLL-Dateien wichtig. Es werden .fter Routinen einer bestimmten Bibliothek neugeschrieben und nicht in der Bibliothek ersetzt, sondern erst beim Erzeugen der .EXE-Datei angegeben. Wird eine .PLL-Datei erzeugt, die eine solche Bibliotheks-Routine referenziert, und .RTLink mu. eine Referenz auf diese Routine aufl.sen, so kennt .RTLink zu diesem Zeitpunkt das neue Modul noch nicht. Daher wird das Modul der Bibliothek in die .PLL-Datei eingebunden. Wird sp.ter die .EXE-Datei erzeugt, die das neue Modul beinhaltet, so erzeugt .RTLink diese Warnung. Denn die .PLL-Datei enth.lt eine Referenz auf die Bibliotheksfunktion und die .EXE-Datei eine Referenz auf die neugeschriebene Routine.

L.sung: Die Redefinition eines Symbols ist normalerweise ein ernsthafter Fehler, nur in einigen F.llen kann diese Meldung ignoriert werden, falls das Symbol nicht angesprochen werden sollte. In den meisten F.llen hat diese Meldungen allerdings katastrophale Auswirkungen, da wahrscheinlich Referenzen auf das Symbol in verschiedenen Programmteilen mit verschiedenen Adressen ersetzt wurden. Durch die EXCLUDE-Option kann diese Meldung h.ufig vermieden werden, besonders dann, wenn die Definition von Symbolen in einer .PLL-Datei verz.gert werden soll, damit die korrekte Definition w.hrend des .EXE Linkvorgangs gefunden wird.

wrt0012 Pre-Linked Library symbol '<Symbol>' redefined

Problem: Diese Meldung ist besonders im Zusammenhang mit .PLL-Dateien wichtig. Es werden .fter Routinen einer bestimmten Bibliothek neugeschrieben und nicht in der Bibliothek ersetzt, sondern erst beim Erzeugen der .EXE-Datei angegeben. Wird eine .PLL-Datei erzeugt, die eine solche Bibliotheks-Routine referenziert, und .RTLink mu. eine Referenz auf diese Routine aufl.sen, so kennt .RTLink zu diesem Zeitpunkt das neue Modul noch nicht. Daher wird das Modul der Bibliothek in die .PLL-Datei eingebunden. Wird sp.ter die .EXE-Datei erzeugt, die das neue Modul beinhaltet, so erzeugt .RTLink diese Warnung. Denn die .PLL-Datei enth.lt eine Referenz auf die Bibliotheksfunktion und die .EXE-Datei eine Referenz auf die neugeschriebene Routine.

L.sung: Die Redefinition eines Symbols ist normalerweise ein ernsthafter Fehler, nur in einigen F.llen kann diese Meldung ignoriert werden, falls das Symbol nicht angesprochen werden sollte. In den meisten F.llen hat diese Meldungen allerdings katastrophale Auswirkungen, da wahrscheinlich Referenzen auf das Symbol in verschiedenen Programmteilen mit verschiedenen Adressen ersetzt wurden. Durch die EXCLUDE-Option kann diese Meldung h.ufig vermieden werden, besonders dann, wenn die Definition von Symbolen in einer .PLL-Datei verz.gert werden soll, damit die korrekte Definition w.hrend des .EXE Linkvorgangs gefunden wird.

CMONTH() Ermittelt den Monatsnamen eines Datums

Syntax

CMONTH(<dDatum>) -> cMonat

Parameter

<dDatum> ist das zu konvertierende Datum.

R.ckgabe

CMONTH() gibt eine Zeichenkette zur.ck, die den Namen des Monats enth.lt. Der erste Buchstabe ist ein Gro.buchstabe, alle weiteren sind Kleinbuchstaben. Die maximale L.nge betr.gt neun Zeichen f.r den Monat September. Ein leerer Datumswert erzeugt eine leere Zeichenkette ("").

Anwendung

CMONTH() ist eine Datumskonvertierungsfunktion, die angewendet werden kann, um formatierte Listen, Etiketten oder Bildschirmausgaben zu erzeugen.

Beispiele

Das Beispiel zeigt die verschiedenen Einsatzm.glichkeiten der Funktion CMONTH():

```
? CMONTH(DATE())                  // Resultat: September
? CMONTH(DATE() + 45)             // Resultat: Oktober
? STR( DAY( DATE())) + "." + ;
SUBSTR(CMONTH(DATE()), 1, 4)     // Resultat: 29. Sept.
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

wrt0014 Cannot define linker reserved symbol '<Symbol>' because it is in the don't define list

---

Problem: Ein reserviertes Symbol, das der Linker ben.tigt, um Adressen korrekt zu erzeugen, wurde durch die EXCLUDE-Option ausgeschlossen. Solche Symbole d.rfen nicht mit EXCLUDE spezifiziert werden.

L.sung: Die folgenden Symbole k.nnen m.glicherweise (je nach Programmstruktur) als reservierte Symbole durch .RTLink eingesetzt werden: _end, _edata, $$OVLBASE, $$MAIN, $$CGSN, $$COVL, $$MPGSNBASE, $$MGPSNOVL, $$MPOVLLFA, $$EXENAM und $$INTNO.

wrt0022 .EXE may not execute properly - undefined symbols

Problem: .RTLink erzeugt diese Warnung, wenn undefinierte Symbole nach dem Linklauf in der .EXE-Datei .briggeblieben sind. Diese Meldung setzt den DOS-R.ckgabewert auf 1.

L.sung: Undefinierte Symbole am Ende der Erzeugung einer .PLL-Datei sind im allgemeinen kein Problem. Wird allerdings eine .EXE-Datei erzeugt, ist das erzeugte Programm normalerweise nicht ablauff.hig. Es ist zu pr.fen, ob alle n.tigen Bibliotheken, .PLL- und .OBJ-Dateien angegeben wurden.

ert0004 Duplicate stacks

Problem: Das Programm fordert zwei verschiedene Stacks an. Der Grund daf.r ist ein Assemblerprogramm, das direkt die Allozierung von Stacksegmenten kontrolliert.

L.sung: Eine der beiden Stackanforderungen mu. entfernt werden. Die Register SS und SP k.nnen im Programm selber ver.ndert werden, der Linker mu. aber ein einzelnes Stacksegment identifizieren, um die Register im .EXE-Header korrekt zu initialisieren.

ert0005 filename portion of output file '<Dateiname>' too long

---

(max length <Maximale L.nge des Ausgabedateinamens>)

Problem: Es wurde ein Dateiname f.r die Ausgabedatei angegeben, der zu lang f.r den Linker ist. Die Meldung zeigt die maximal m.gliche L.nge f.r den Dateinamen an.

L.sung: Die L.nge des Dateinamens mu. auf die angegebene L.nge reduziert werden. Es ist zu beachten, da. nur die ersten acht Buchstaben f.r einen DOS-Dateinamen signifikant sind.

ert0010 Overlay Area Nesting Area

Problem: Es wird die FREEFORMAT-Schnittstelle genutzt, und es existiert ein ENDAREA-Statement ohne dazugeh.riges BEGINAREA-Statement.

L.sung: Die Eingabedatei mu. auf dieses Problem hin .berpr.ft werden.

ert0011 PRELOAD command must have active section

Problem: Das Kommando FREEFORMAT PRELOAD definiert, da. ein Overlay-Bereich bereits w.hrend des Programm-Ladevorganges geladen werden soll. Das Kommando mu. nach der SECTION-Anweisung f.r das entsprechende Overlay stehen. Dieser Fehler weist darauf hin, da. der Root-Bereich durch das PRELOAD-Statement geladen werden soll. Dieser Bereich wird jedoch immer bei Programmstart geladen und nie durch den Overlaymanager verwaltet.

L.sung: Das PRELOAD-Kommando mu. korrekt plaziert werden.

ert0013 Input Terminated with area left open

Problem: Die FREEFORMAT-Beschreibung wird genutzt, und es existiert ein BEGINAREA-Statement ohne dazugeh.riges ENDAREA-Statement.

L.sung: Die Eingabedatei mu. auf dieses Problem hin .berpr.ft werden.

ert0014 Overlays not supported in Pre-Linked Library

Problem: Wird eine .PLL-Datei erzeugt, darf nur ein Rootbereich existieren. Der gesamte Code der .PLL-Datei mu. Teil des Rootbereiches sein und kann nicht in Overlays unterteilt werden.

L.sung: Wird die FREEFORMAT-Beschreibung zusammen mit der PRELINK-Option genutzt, darf keine BEGINAREA-, ENDAREA- oder SECTION-Option angegeben werden.

ert0015 Bad format binary number '<Zahl>'

Problem: Es wurde eine ung.ltige Zahl f.r eine Option oder ein Kommando eingegeben, das eine Bin.rzahl als Eingabe erwartet.

L.sung: Nur 0 und 1 sind g.ltige Bin.rziffern.

ert0016 Bad format octal number '<Zahl>'

Problem: Es wurde eine ung.ltige Zahl f.r eine Option oder ein Kommando eingegeben, das eine Octalzahl als Eingabe erwartet.

L.sung: Nur die Ziffern von 0 bis 8 sind f.r Octalzahlen g.ltig.

ert0017 Bad format decimal number '<Zahl>'

Problem: Es wurde eine ung.ltige Zahl f.r eine Option oder ein Kommando eingegeben, das eine Dezimalzahl als Eingabe erwartet.

L.sung: Nur die Ziffern von 0 bis 9 sind f.r Dezimalzahlen g.ltig.

ert0018 Bad format hexacdecimal number '<Zahl>'

Problem: Es wurde eine ung.ltige Zahl f.r eine Option oder ein Kommando eingegeben, das eine Hexadezimalzahl als Eingabe erwartet.

L.sung: Nur die Ziffern von 0 bis 9 und die Buchstaben A bis F sind f.r Hexadezimalzahlen g.ltig.

ert0024 Cannot open .RTLink configuration file '<Dateiname>'

Der Linker hat die angegebene-Datei gefunden, konnte sie aber (z.B. wegen Datei- oder Festplattenfehler) nicht .ffnen.

ert0025 Cannot open script file '<Dateiname>'
Problem: Der Linker kann die angegebene Script-Datei (durch ein '@' in der Kommandozeile gekennzeichnet) nicht finden.

L.sung: .RTLink benutzt keine Environment-Variablen, um nach Script-Dateien zu suchen, daher mu. der Dateiname Laufwerks- und Pfadbezeichnungen sowie die Dateinamenserweiterung beinhalten. Korrigieren Sie den Dateinamen in der Kommandozeile entsprechend.

ert0026 Output Name not specified or implied

Problem: Der Linker wei. nicht, in welche Datei er seine Ausgabe schreiben soll.

L.sung: Ein Namen f.r die Ausgabedatei ist explizit zu spezifizieren.

ert0028 Cannot find transfer file '<Dateiname>'

Problem: Die PLL-Option wurde benutzt, und die damit verbundene Transferdatei (.PLT) mit dem entsprechenden Dateinamen konnte nicht gefunden werden. Falls der Dateiname inklusive Pfad korrekt angegeben wurde, konnte die Datei nicht ge.ffnet werden, andernfalls wurde die Datei im aktuellen Verzeichnis, sowie in den durch die Environment-Variablen PLL und LIB spezifizierten Verzeichnissen gesucht und nicht gefunden.

L.sung: Pr.fen Sie, ob der Dateiname f.r die .PLL-Datei korrekt angegeben wurde und eine entsprechende .PLT-Datei existiert.

ert0029 Cannot Open Transfer File '<Dateiname>'

Der Linker hat die angegebene Datei gefunden, konnte sie aber (z.B. wegen Datei- oder Festplattenfehler) nicht .ffnen.

ert0030 Disallowed symbol '<Symbol>' already loaded in .PLL '<PLL Dateiname>'
Problem: Ein Symbol, das mit der Option EXCLUDE spezifiziert wurde, ist bereits in einer angesprochenen .PLL-Datei geladen. Der Linker kann daher das gegebene Symbol nicht ausschlie.en.

L.sung: Benutzen Sie die EXCLUDE-Option f.r dieses Symbol, wenn die .PLL-Datei erzeugt wird.

ert0032 User-level PLL transferred to .RTLink producing PLL

Problem: Im Prelink-Modus wurde eine .PLL-Datei angegeben, die auf einer anderen .PLL-Datei aufbaut.

L.sung: Es ist m.glich, durch die Angabe der beiden Optionen /PRELINK und /PLL eine .PLL-Datei in Abh.ngigkeit von einer anderen .PLL-Datei zu erzeugen. Diese Verschachtelung kann aber nur eine Ebene weit gehen. D.h., wenn B.PLL mit Hilfe von A.PLL erzeugt wurde, kann C.PLL nicht mit B.PLL erzeugt werden.

ert0033 Invalid format in transfer (.PLT) file '<Dateiname>'

(**possi**bly caused by a bad .PLL/PLT name)
Problem: Entweder wurde eine ung.ltige .PLT-Datei angegeben, oder die Datei ist fehlerhaft.

L.sung: Zuerst sollte der Dateiname .berpr.ft werden. Sollte der richtige Dateiname spezifiziert sein, mu. die .PLT neu erzeugt werden, indem die dazugeh.rige .PLL-Datei neu generiert wird.

ert0035 Cannot find object file '<Dateiname>'

Problem: Die angegebene Objekt-Datei (.OBJ) konnte nicht gefunden werden. Falls der Dateiname inklusive Pfad korrekt angegeben wurde, konnte die Datei nicht ge.ffnet werden. Andernfalls wurde die Datei im aktuellen Verzeichnis, sowie in den durch die Environment-Variablen OBJ und LIB spezifizierten Verzeichnissen gesucht und nicht gefunden.

L.sung: Pr.fen Sie, ob die Objekt-Datei existiert, bzw. der korrekte Name angegeben wurde.

ert0041 Unrecognized option '<Option>'

Problem: Eine ung.ltige POSITIONAL-Option wurde angegeben.

L.sung: Die ung.ltige Option mu. entfernt werden. Eine Liste g.ltiger Optionen finden Sie im Kapitel Der Clipper Linker - RTLINK.EXE.

ert0044 Illegal STACK size

Problem: Ein ung.ltiger Parameter wurde bei der STACK-Option angegeben.

L.sung: Eine korrekte Stackgr..e mu. angegeben werden. Im Kapitel Der Clipper Linker - RTLINK.EXE finden Sie die korrekte Syntax f.r die STACK-Option.

ert0045 Cannot create transfer file '<Dateiname>'

Problem: Als die angegebene Datei erstellt werden sollte, trat ein .ffnungsfehler auf. Normalerweise tritt ein solcher Fehler auf, wenn der Zugriff verweigert wird oder die Pfadangabe nicht korrekt ist.

L.sung: .ndern Sie den Namen der .PLL-Datei, die erzeugt werden soll.

ert0048 Only one Output Name should be specified

Problem: Es wurde mehr als ein Ausgabe-Name am POSITIONAL "Output Name" Prompt spezifiziert.

L.sung: Geben Sie nur einen Ausgabe-Namen an.

ert0049 Cannot create map file '<map>'

Problem: Der Linker kann die angegebene Map-Datei nicht erzeugen. Normalerweise liegt das an einem ung.ltigen Dateinamen, einem Schreibschutz auf einer Datei gleichen Namens oder einem nicht existierenden Verzeichnis als Teil des angegebenen Dateinamens.

L.sung: Korrigieren Sie den Dateinamen f.r die Map-Datei.

ert0052 Attempt to load module defining symbol '<Symbol>'

Problem: Der Linker versuchte, ein Modul zu laden, welches das spezifizierte Modul mit der EXCLUDE-Option definiert.

L.sung: Diese Nachricht erscheint, wenn ein Symbol mit EXCLUDE angegeben wurde, aber in einer der gelinkten Objekt-Dateien definiert wird. Dies ist nicht erlaubt. Entweder geh.rt eine der Objekt-Dateien nicht in diesen Linklauf, oder das Symbol nicht in die EXCLUDE-Liste.

ert0054 Cannot find library file '<Bibliothek>'

Problem: Der Linker kann die angegebene Bibliothek nicht finden. Falls der Dateiname inklusive Pfad korrekt angegeben wurde, konnte die Datei nicht ge.ffnet werden, andernfalls wurde die Datei im aktuellen Verzeichnis, sowie in den durch die Environment-Variable LIB spezifizierten Verzeichnissen gesucht und nicht gefunden.

L.sung: Pr.fen Sie, ob die Bibliothek existiert und der korrekte Name angegeben wurde.

ert0055 Cannot seek within library dictionary

Problem: Der Linker hat einen Fehler bei der Bearbeitung der aktuellen Bibliothek entdeckt. Eine Bibliothek besitzt ein erweitertes Verzeichnis (Extended dictionary) am Ende der Datei, aber die Suche nach diesem Verzeichnis ist fehlgeschlagen. Normalerweise ist eine solche Bibliothek zerst.rt.

L.sung: Lassen Sie CHKDSK /F laufen, um Laufwerksfehler festzustellen. Ist das nicht der Fall, erzeugen oder installieren Sie die Bibliothek neu (Durch die VERBOSE-Option l..t sich die fehlerhafte Bibliothek feststellen). Tritt der Fehler weiterhin auf, kann er mit der Option NOEXTDICTIONARY umgangen werden.

ert0056 Read of random library table failed

L.sung: Lassen Sie CHKDSK /F laufen, um Laufwerksfehler festzustellen. Ist dies nicht der Fall, erzeugen oder installieren Sie die Bibliothek neu (Durch die VERBOSE-Option l..t sich die fehlerhafte Bibliothek feststellen).

ert0061 Input/ouput error on output file <Dateiname>
Problem: Der Linker kann die angegebene Datei nicht lesen und/oder nicht in sie schreiben.

L.sung: Da diese Meldung normalerweise auf eine korrupte Datei oder ein korruptes Laufwerk hinweist, sollten Sie versuchen, diesen Fehler mit CHKDSK /F zu beseitigen.

ert0066 Write to map file <Dateiname> failed

Problem: Ein Schreibversuch in die angegebene Map-Datei ist fehlgeschlagen. Wenn diese Meldung auftritt, ist die .EXE- oder .PLL-Datei bereits erzeugt worden. Nur die Map-Datei wird fehlerhaft sein, da sie nicht komplett weggeschrieben werden konnte.

L.sung: L.schen Sie .berfl.ssige Dateien auf der Festplatte, um den n.tigen Plattenspeicherplatz freizumachen. Starten Sie dann einen erneuten Linklauf.

ert0067 Not enough characters in command '<Option>' to guarantee
---

uniqueness
Problem: Eine Zeichenfolge, die in fr.heren Versionen eine bestimmte Option spezifiziert hat, ist nicht mehr lang genug, um eine eindeutige Identifizierung der gew.nschten Option zu erm.glichen.

L.sung: Die minimale L.nge der Abk.rzung einer jeden Option ist im Kapitel Der Clipper Linker - RTLINK.EXE dokumentiert.

ert0077 Cannot find startup code 'RTLINKST.COM'

Problem: W.hrend eine .PLL-Datei durch die PRELINK-Option erzeugt wird, konnte die Datei RTLINKST.COM nicht gefunden werden.

L.sung: RTLINKST.COM enth.lt den n.tigen start-up Code, der in einer .PLL-Datei enthalten sein mu.. Wenn diese Datei ben.tigt wird, durchsucht .RTLink nur das Verzeichnis, in dem sich RTLINK.EXE befindet.

COL() Ermittelt die Spaltenposition des Bildschirm-Cursors

Syntax

COL() -> nSpalte

R.ckgabewert

COL() erzeugt einen numerischen ganzzahligen Wert, der zwischen 0 und MAXCOL() liegt.

Anwendung

COL() ist eine Bildschirmfunktion und gibt die derzeitige Spaltenposition des Cursors auf dem Bildschirm zur.ck. Der Wert von COL() .ndert sich immer dann, wenn sich die Spaltenposition des Cursors .ndert. Sowohl Konsolenbefehle (?, ??) als auch Bildschirmbefehle k.nnen die Spaltenposition des Cursors .ndern. Die Befehle CLEAR und CLEAR SCREEN setzen COL() automatisch auf Null zur.ck.

COL() kann dazu dienen, den Cursor relativ zur aktuellen Spaltenposition zu bewegen. Die haupts.chliche Anwendung besteht in der Kombination mit ROW() und den verschiedenen M.glichkeiten der @... Befehle. COL() und ROW() k.nnen im Einzelfall auch dazu benutzt werden, cursorpositions-abh.ngige Prozeduren oder Funktionen zu entwickeln, um zum Beispiel das aktuelle Eingabefeld nicht zu .berschreiben.

Ist die Ausgabeneinheit mit SET DEVICE TO PRINTER auf den Drucker gesetzt, werden mit den @...SAY-Befehlen PROW() und PCOL() beeinflu.t, ROW() und COL() bleiben unber.cksichtigt.

Beispiele

In diesem Beispiel wird ab Spalte 10 eine Zeichenfolge und der Name eines Kunden ausgegeben. Nachfolgende Leerzeichen in der Namensangabe werden mit der Funktion TRIM() entfernt, um zwei Spalten weiter den Kontostand des Kunden auszugeben:

```
USE Verkauf NEW
@ 1, 10 SAY "Name des Kunden: " + TRIM(Name)
@ 1, COL() + 2  SAY "Kontostand: " + Saldo
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

ert0078 Error reading RTLINKST.COM

Problem: .RTLink konnte RTLINKST.COM finden, allerdings ist ein Lesefehler aufgetreten.

L.sung: M.glicherweise ist die Datei korrupt. Installieren Sie RTLINKST.COM erneut von Ihren Disketten.

ert0083 Cannot find RTLINK.DAT (freeformat parser data)

Problem: .RTLink konnte die Datei RTLINK.DAT nicht finden.

L.sung: RTLINK.DAT wird bei jedem Aufruf von RTLINK.EXE ben.tigt. Dabei durchsucht der Linker nur das Verzeichnis, in dem sich RTLINK.EXE befindet.

ert0084 Cannot read RTLINK.DAT header (freeformat parser data)

Problem: .RTLink konnte RTLINK.DAT finden, allerdings ist ein Lesefehler aufgetreten.

L.sung: M.glicherweise ist die Datei korrupt, installieren Sie RTLINK.DAT erneut von Ihren Disketten.

ert0085 Freeformat parser tables too big to fit in memory

Problem: Es ist nicht gen.gend Speicher vorhanden, um die Parser-Tabelle aus RTLINK.DAT zu laden.

L.sung: Damit .RTLink korrekt arbeiten kann, mu. der verf.gbare Hauptspeicher vergr..ert werden.

ert0086 Cannot read RTLINK.DAT data (freeformat parser data)

Problem: .RTLink konnte RTLINK.DAT finden, allerdings ist ein Lesefehler aufgetreten.

L.sung: M.glicherweise ist die Datei korrupt. Installieren Sie RTLINK.DAT erneut von Ihren Disketten.

ert0088 Illegal information level <Stufe> requested

Problem: Es wurde eine Informationsstufe mit VERBOSE angegeben, die nicht zwischen 0 und 2 liegt.

L.sung: Geben Sie eine korrekte Informationsstufe an.

ert0092 Cannot find RTLINK.HLP

Problem: Die HELP-Option wurde angegeben, aber .RTLink kann die Datei RTLINK.HLP nicht finden.

L.sung: Wird RTLINK.HLP ben.tigt, so sucht der Linker nur in dem Verzeichnis, in dem sich auch RTLINK.EXE befindet.

ert0097 Invalid MAP flag <Typ>

Problem: Die MAP-Option wurde mit einem ung.ltigen Schalter spezifiziert.

L.sung: S, N, und A sind g.ltige Schalter f.r die MAP-Option.
Erkl.rungen hierzu finden Sie im Kapitel Der Clipper Linker - RTLINK.EXE.

ert0112 File name <Dateiname> is more than <L.nge> characters long

---

Problem: Die L.nge des angegebenen Dateinamens (inklusive Pfad) .berschreitet die maximal zul.ssige Gr..e.

L.sung: Normalerweise weist diese Meldung auf einen Syntaxfehler in der Eingabe f.r .RTLink hin.

eut0003 Not enough free memory

Problem: Obwohl .RTLink auch in Systemen mit wenig Hauptspeicher lauff.hig ist, gibt es ein Minimum an verf.gbaren Speicher. Wenn .RTLink in den Speicher geladen worden ist, m.ssen noch ca. 10 KB RAM verf.gbar sein, damit der Linker interne Informationen .ber Symbole und Code speichern kann.

L.sung: Versuchen Sie, den verf.gbaren Hauptspeicher zu vergr..ern. Entfernen Sie dazu zum Beispiel TSR-Programme aus dem Speicher.

eut0006 Need to swap but nothing is on the swap list

Problem: Es ist nicht gen.gend freier Speicher f.r den Linklauf vorhanden. .RTLink versucht in solchen Situationen, Daten aus dem Hauptspeicher in eine Swap-Datei auszulagern. Allerdings m.ssen aktuell ben.tigte Daten im Hauptspeicher bleiben. Diese Nachricht bedeutet, da. nicht gen.gend Speicher vorhanden ist, um diese minimalen Informationen im Speicher zu belassen.

L.sung: Versuchen Sie, den verf.gbaren Hauptspeicher zu vergr..ern.

eut0007 Cannot open work file '<Dateiname>'

Problem: .RTLink kann die angegebene, tempor.re Arbeitsdatei nicht .ffnen. Normalerweise liegt das an einer fehlenden Berechtigung, um in die Datei oder das entsprechende Verzeichnis zu schreiben bzw. aus ihr zu lesen. Es k.nnen auch andere Laufwerksprobleme aufgetreten sein, diese sind allerdings seltener.

L.sung: Stellen Sie sicher, da. Sie Schreib- und Leserechte f.r die Datei und das Verzeichnis besitzen.

eut0009 Write of <Zahl> chars to work file <Dateiname> at position

---

<Position>
returns <Zahl>
Problem: .RTLink kann nicht an die spezifizierte Position in der Tempor.r-Datei schreiben. Normalerweise deutet dieser Fehler auf ungen.genden Plattenplatz hin. Es k.nnen aber auch andere Laufwerksprobleme aufgetreten sein, diese sind allerdings seltener.

L.sung: Sie besitzen ausreichende Rechte, um die tempor.re Datei zu lesen und in diese zu schreiben. Versuchen Sie gegebenenfalls, zus.tzlichen Platz auf Ihrem Laufwerk bereitzustellen. Es m.ssen mindestens Zahl + Position Bytes frei sein.

eut0015 Write to buffered output file '<Dateiname>' at position

---

0x<Position>
(<Zahl> chars) failed
Problem: .RTLink kann nicht an die spezifizierte Position in der Ausgabedatei schreiben. Normalerweise deutet dieser Fehler auf ungen.genden Plattenplatz hin. Es k.nnen aber auch andere Laufwerksprobleme aufgetreten sein, diese sind allerdings seltener.

L.sung: Versuchen Sie gegebenenfalls, zus.tzlichen Platz auf Ihrem Laufwerk bereitzustellen.

eut0016 Open of buffered output file '<Dateiname>' failed

Problem: .RTLink kann die Ausgabedatei nicht .ffnen. Diese Datei wird mit Schreibrechten ge.ffnet. Dieser Fehler weist also darauf hin, da. keine Schreibrechte f.r die Datei oder das Verzeichnis, vorhanden sind (Netzwerke!). Es k.nnen auch andere Laufwerksprobleme aufgetreten sein, diese sind allerdings seltener.

L.sung: Stellen Sie sicher, da. Sie Schreibrechte f.r die Datei und das Verzeichnis besitzen. Falls dies nicht m.glich ist, w.hlen Sie einen anderen Dateinamen, oder ein anderes Verzeichnis, f.r das Sie Schreibrechte besitzen.

eut0019 Bad format object or library file '<Dateiname>'

Problem: Die angegebene Datei besitzt nicht das korrekte Format.

L.sung:

RMAKE Warnungs-Meldungen

RMAKE Ausf.hrungsfehler-Meldungen

Schwerwiegende RMAKE-Fehlermeldungen

RMAKE-Fehlermeldungen -- Allgemein

Dieser Abschnitt ist eine Zusammenfassung der m.glichen Fehlermeldungen, die auftreten k.nnen, wenn das Clipper Make-Programm RMAKE eingesetzt wird. Die Fehler sind entsprechend ihrer Schwere und der Stelle, an der sie in einem Make-Proze. auftreten k.nnen, in verschiedene Kategorien unterteilt. Im folgenden wird jede dieser Kategorien zuerst beschrieben, und dann eine .bersicht aller m.glichen Fehlermeldungen f.r jede Kategorie gegeben. Spezielle Informationen .ber die Benutzung und die Syntax von RMAKE finden Sie im Kapitel Hilfen bei der Programmerstellung - RMAKE.EXE innerhalb dieses Handbuches.

RMAKE-Warnungen

RMAKE Warnungen zeigen potentielle Fehler in einer Make-Datei an, unterbrechen die Bearbeitung derselben jedoch nicht. Der Return-Code von DOS bleibt durch solche Warnungen unbeeinflu.t.
Das generelle Format einer RMAKE Warnung ist:

<Dateiname>(<Zeile>) Warning RMAKE/R1xxx <text>[: <Symbol>]

Falls eine Warnung in keiner direkten Verbindung mit einer Zeile der Make-Datei steht, werden kein Dateiname und keine Zeilennummer angezeigt. Wenn die Option /W nicht angegeben wurde, werden keinerlei Warnungen ausgegeben.

RMAKE-Ausf.hrungsfehler

Unter RMAKE wird dann ein Ausf.hrungsfehler erzeugt, wenn eines der Programme, die in einer der Anweisungszeilen ausgef.hrt wurden, einen DOS Return-Code ungleich 0 liefert. Sobald es zu einem solchen Ausf.hrungsfehler kommt, setzt RMAKE seinen eigenen DOS Return-Code auf den Wert 1, und beendet die Ausf.hrung der Make-Datei, falls nicht die Option /I angegeben wurde.
Das generelle Format eines RMAKE Ausf.hrungs-Fehlers ist:

<Dateiname>(<Zeile>) Error RMAKE/R2xxx Exit <n>: <Kommandozeile>

Die Angabe Exit <n> gibt dabei den DOS Return-Code an, welcher von dem jeweiligen Programm innerhalb der Make-Datei zur.ckgegeben wurde.

Schwerwiegende RMAKE Fehlermeldungen

Schwerwiegende Fehler weisen auf ernsthafte Probleme innerhalb einer Make-Datei hin, die von RMAKE nicht behoben oder umgangen werden k.nnen. Solche Fehler k.nnen sowohl w.hrend der Make- als auch w.hrend der Parser-Phase auftreten. Bei einem schwerwiegenden Fehler stoppt RMAKE die Ausf.hrung der Make-Datei und liefert den DOS Return-Code 1.
Das generelle Format eines schwerwiegenden Fehlers unter RMAKE lautet:
<Dateiname>(<Zeile>) Fatal RMAKE/R3xxx <text>[: <Symbol>]

Falls eine Warnung in keiner direkten Verbindung mit einer Zeile der Make-Datei steht, werden kein Dateiname und keine Zeilennummer angezeigt.

R1001 Ignoring redefinition of command-line macro

Problem: In einer Make-Datei wurde versucht, ein Makro umzudefinieren, das zuvor in der Kommandozeile oder in der Umgebungsvariablen RMAKE definiert worden ist. Der Versuch wird ignoriert und das Makro beh.lt seinen Wert.

L.sung:b Die Direktive #undef kann benutzt werden, um eine Makro-Definition aus der Kommandozeile aufzuheben, um dieser dann einen neuen Wert zuzuweisen.

R1002 Target does not exist: '<Datei>'

Problem: Die angezeigte Zieldatei wurde innerhalb einer Abh.ngigkeits-Regel angegeben und existiert noch nicht.

L.sung: RMAKE wird die Zieldatei erzeugen.

R1003 Ignoring text: '<Text>'

Problem: Eine Anweisung in einer Make-Datei enth.lt nicht dazugeh.rige Zeichen. Diese Zeichen werden ignoriert.

L.sung: Untersuchen Sie die Make-Datei, um festzustellen, ob die nicht dazugeh.rigen Zeichen nicht das Resultat eines anderen Fehlers sind. Falls nicht, l.schen Sie diese Zeichen aus der Make-Datei.

R2001 Exit n: '<Anweisungszeile>'

Problem: Ein DOS Return-Code ungleich 0 wurde von einem der Programme zur.ckgegeben, das als Anweisung in einer Abh.ngigkeits- oder Schlu.folgerungs-Regel angegeben ist. Der Make-Durchlauf wird beendet, es sei denn, die Option /I ist verwendet worden.

L.sung: Das angezeigte Programm wurde aus irgendeinem Grunde nicht normal beendet, eventuell durch eine inkorrekte Script-Datei oder eine v.llig fehlende Datei. Dieser Fehler mu. behoben werden, bevor der Make-Proze. erneut gestartet wird.

R3001 Too many make files

Problem: Es wurde versucht, mehr als 16 Make-Dateien innerhalb der Kommandozeile oder der Umgebungsvariablen RMAKE anzugeben.

L.sung: Reduzieren Sie die Anzahl der Make-Dateien.

R3002 Can't open: '<Datei>'

Problem: RMAKE konnte eine Make-Datei, die in der Kommandozeile, in der Umgebungsvariablen RMAKE oder in einer #include-Direktive angegeben wurde, nicht .ffnen.

L.sung: Ein vollst.ndiger oder teilweiser Pfad kann angegeben werden, um RMAKE so zu veranlassen, im angegebenen Pfad bzw. auf dem angegebenen Laufwerk nach der Datei zu suchen. Wird keinerlei Pfad angegeben, sucht RMAKE nur im angemeldeten Verzeichnis. Bei Verwendung der Option /S werden allerdings alle Unterverzeichnisse des angemeldeten oder angegebenen Verzeichnisses mit durchsucht.

R3003 Invalid option: '<Option>'

Problem: In der Kommandozeile oder in der Umgebungsvariablen RMAKE wurde eine unzul.ssige Option angegeben.

L.sung: Eine .bersicht der zul.ssigen Optionen ist im Kapitel Hilfen bei der Programmerstellung - RMAKE.EXE in diesem Handbuch zu finden.

R3004 Out of memory

Problem: Es ist kein Arbeitsspeicher f.r weitere Definitionen verf.gbar.

L.sung: Der Make-Proze. sollte vereinfacht werden, indem er zum Beispiel zweigeteilt wird.

R3005 Internal workspace exhausted

Problem: Der interne Arbeitsbereich ist ersch.pft.

L.sung: Dieser Bereich kann .ber die Option /XW vergr..ert werden.

R3006 Symbol table exhausted

Problem: Die interne Symboltabelle ist ersch.pft.

L.sung: Diese Tabelle kann .ber die Option /XS vergr..ert werden.

R3007 String too large

Problem: Eine einzelne Zeichenkette war l.nger als 1024 Byte.

L.sung: Vereinfachen Sie die entsprechende Zeile in der Make-Datei.

R3008 String table exhausted

Problem: Kein weiterer Speicher f.r Definitionen verf.gbar.

L.sung: Der Make-Proze. sollte vereinfacht werden, indem er zum Beispiel zweigeteilt wird.

R3009 File table exhausted

Problem: Der Speicherbereich f.r die Tabelle, in der Dateinamen und deren Datum bzw. Uhrzeit verwaltet werden, ist ersch.pft.

L.sung: Der Make-Proze. sollte vereinfacht werden, indem er zum Beispiel zweigeteilt wird.

R3010 Too many actions

Problem: In einer Abh.ngigkeits- oder Schlu.folgerungs-Regel wurden entweder mehr als 32 Anweisungen angegeben, oder die Gesamtl.nge eines Anweisungsblockes .berschreitet 1024 Byte.

L.sung: Vereinfachen Sie die betroffene Definition.

R3011 Too many dependencies

Problem: In einer Abh.ngigkeits-Regel wurden mehr als 128 abh.ngige Dateien angegeben.

L.sung: Vereinfachen Sie die Regel.

R3012 Syntax error: '<Token>'

Problem: Innerhalb der angezeigten Regel oder Direktive ist ein Syntax-Fehler aufgetreten.

L.sung: Die korrekte Syntax f.r alle Regeln und Direktiven in einer Make-Datei kann im Kapitel Hilfen bei der Programmerstellung - RMAKE.EXE in diesem Handbuch gefunden werden.

R3013 Unbalanced parentheses

Problem: Eine Regel oder Direktive enth.lt ein Klammern-Ungleichgewicht.

L.sung: Zu jeder .ffnenden Klammer mu. eine schlie.ende Klammer vorhanden sein. Suchen Sie die betroffene Zeile in der Make-Datei und beheben Sie den Fehler.

R3014 #else without if

Problem: Eine #else-Direktive wurde gefunden, ohne da. sich davor eine zugeh.rige #if-Direktive befand.

L.sung: Ein #else mu. sich in einer Make-Datei immer zwischen einem #if und einem #endif befinden - korrigieren Sie den Fehler entsprechend.

R3015 #endif without if

Problem: Eine #endif-Direktive wurde gefunden, ohne da. sich davor eine zugeh.rige #if-Direktive befand.

L.sung: Ein #endif ist in einer Make-Datei nur zul.ssig, wenn ihm ein #if vorausgegangen ist - korrigieren Sie den Fehler entsprechend.

R3016 Open conditionals

Problem: Am Ende einer Make-Datei sind ein oder mehrere #if-Direktiven nicht .ber ein #endif abgeschlossen worden.

L.sung: Pr.fen Sie die Make-Datei sorgf.ltig und schlie.en Sie jedes #if-Konstrukt durch ein #endif. Die Benutzung von Einr.ckungen f.r die Konstrukte einer Make-Datei macht eine solche Aufgabenstellung wesentlich leichter.

R3017 Unrecognized directive: '<Direktive>'

Problem: Eine unzul.ssige Direktive wurde gefunden.

L.sung: Eine komplette Liste der Direktiven f.r eine Make-Datei finden Sie im Kapitel Hilfen bei der Programmerstellung - RMAKE.EXE in diesem Handbuch.

R3018 Dependency does not exist: '<Datei>'

Problem: Eine Abh.ngigkeits-Regel hat eine nicht vorhandene Datei als abh.ngig angegeben. RMAKE kann nicht feststellen, ob die Zieldatei .ber ein .lteres Datum oder eine .ltere Uhrzeit verf.gt.

L.sung: RMAKE sucht normalerweise nur im angemeldeten Verzeichnis nach abh.ngigen Dateien, es sei denn, Sie verwenden innerhalb der Make-Datei ein spezielles Makro (makepath-Makro), .ber das zus.tzliche Pfade zur Suche solcher Dateien angegeben werden k.nnen. Weitere Informationen .ber dieses spezielle Makro finden Sie im Kapitel Hilfen bei der Programmerstellung - RMAKE.EXE in diesem Handbuch.

R3019 Circular dependency

Problem: Es wurde festgestellt, da. eine Zieldatei direkt oder indirekt .ber mehrere Abh.ngigkeits-Regeln hinweg von sich selbst abh.ngig ist.

L.sung: Circul.re Abh.ngigkeiten sind nicht erlaubt. Pr.fen Sie die Make-Datei und entfernen Sie derartige Abh.ngigkeiten.

R3020 Environment overflows workspace

Problem: Ein oder mehrere DOS Set-Kommandos wurden als Anweisungen f.r eine Regel angegeben und der Text dieser Kommandos hat die Grenze des internen Arbeitsbereiches (Workspace) .berschrieben.

L.sung: Der interne Arbeitsbereich kann .ber die Option /XW vergr..ert werden.

R3021 Error in redirection

Problem: Bei dem Versuch, die Ausgaben in einer Anweisungszeile an eine oder aus einer Einheit (Datei) umzulenken, ist ein Fehler aufgetreten.

L.sung: Diese Fehlermeldung weist in der Regel darauf hin, da. eine Datei f.r die Umlenkung nicht angelegt oder nicht ge.ffnet werden konnte, weil zum Beispiel kein Platten-Platz mehr verf.gbar war oder eine existierende Datei versehentlich gel.scht wurde.

R3022 Can't execute '<Anweisungszeile>'

Problem: Bei dem Versuch, eine Anweisungszeile auszuf.hren, ist ein Fehler aufgetreten.

L.sung: Pr.fen Sie die Anweisungszeile, um festzustellen, ab die Syntax, die f.r einen Programm-Aufruf benutzt wurde.

CTOD() Wandelt eine Datumszeichenfolge in einen Datumswert um

---

Syntax

CTOD(<cDatum>) -> dDatum

Parameter

<cDatum> ist eine Zeichenfolge, die durch Zahlen den Tag, den Monat und das Jahr repr.sentiert, die jeweils durch ein Trennzeichen getrennt sind. Das Trennzeichen darf jedes beliebige Zeichen au.er einer Zahl sein. Beachten Sie dabei, da. die Reihenfolge von Tag, Monat und Jahr von der Anweisung SET DATE abh.ngt. Werden die Jahrhunderte nicht mit angegeben, so bezieht sich die Datumsangabe auf die Einstellung von SET EPOCH.

R.ckgabe

Es wird ein Wert vom Typ Datum zur.ckgegeben.

Anwendung

CTOD() ist eine Funktion der Zeichenkonvertierung, welche das Datum einer Zeichenfolge in der Reihenfolge, die mit der SET DATE-Anweisung eingestellt worden ist, ermittelt. Der Standardwert ist AMERICAN mm/tt/jj. Um einen leeren Datumswert zu initialisieren, geben Sie eine leere Zeichenkette (""), SPACE(8) oder " / / " an. Ist der angegebene Wert kein g.ltiges Datum, liefert CTOD() einen leeren Datumswert.

CTOD() kann immer dann benutzt werden, wenn eine Zeichenfolge als Datum verwendet werden soll:

. Initialisierung einer Variablen vom Typ Datum
. Als RANGE-Bedingung einer @...Get-Anweisung
. Rechnen mit Datumsangaben, die als Zeichenfolgen vorliegen
. Vergleich von Datumsoperationen mit Zeichenfolgen
. Ablegen einer Datumszeichenfolge in einem Datums-Datenbankfeld

CTOD() ist die umgekehrte Funktion von DTOC(), die einen Datumstyp in eine Zeichenfolge umwandelt und dabei die Einstellungen von SET DATE und SET CENTURY ber.cksichtigt. Die Funktion DTOS() konvertiert einen Datumswert in die Zeichenfolge jjjjmmtt.

Hinweise

. Jahrhundert: Werden nur zwei Stellen f.r das Jahr angegeben, so wird die Einstellung von SET EPOCH ber.cksichtigt. Standardeinstellung von SET EPOCH ist das zwanzigste Jahrhundert.

. Leere Datumsangaben: Um ein leeres Datum anzugeben, benutzen Sie eine der folgenden Formen: SPACE(8), "" oder " / / " als Funktionsargument.

Beispiele

. Das folgende Beispiel initialisiert zwei Datumsvariablen und benutzt die eine in einem GET-Feld und die andere in der RANGE-Bedingung:

```
SET CENTURY ON
SET DATE GERMAN
dStart  := CTOD("27.01.1952")
dEing   := CTOD("")
@ 10, 10 SAY "Datum : " GET dEing RANGE dStart, DATE()
READ
```

. Das n.chste Beispiel benutzt CTOD(), um in einer REPLACE-Anweisung mit Bedingung einen Datumswert zu erzeugen:

```
USE Inventar IN 0
REPLACE ALL Preis WITH Preis * 1.1 FOR Einkauf < CTOD("10.10.89")
```

. Ein weiteres Beispiel zeigt die unterschiedlichen Ergebnisse von CTOD() durch die Einstellungen mit SET EPOCH und SET DATE:

```
? CTOD("01/03/52")    // Resultat: 03.01.1952
```

```
SET DATE TO GERMAN
? CTOD("01/03/52")     // Resultat: 01.03.1952
SET EPOCH TO 2000
? CTOD("01/03/52")     // Resultat: 01.03.2052
? CTOD("01/03/1952")   // Resultat: 01.03.1952
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

CURDIR() Ermittelt das aktuelle DOS-Verzeichnis

Syntax

CURDIR([<cLaufwerk>]) -> cVerzeichnis

Parameter

<cLaufwerk> gibt den Laufwerksbezeichner an, zu dem das aktuelle Verzeichnis ermittelt werden soll. <cLaufwerk> ist vom Typ Zeichen und kann ohne den Doppelpunkt angegeben werden ("A","B",...). Der Vorgabewert ist das aktuelle Laufwerk.

R.ckgabe

CURDIR() gibt das aktuelle Verzeichnis des angegebenen Laufwerks als Zeichenkette ohne f.hrenden \ zur.ck. Tritt bei der Verzeichnisermittlung auf dem Laufwerk ein Fehler auf oder befinden Sie sich im Hauptverzeichnis (ROOT), wird eine leere Zeichenkette ("") zur.ckgegeben.

Anwendung

CURDIR() ist eine Umgebungsfunktion und liefert das aktuell eingestellte DOS-Verzeichnis zur.ck. Die Einstellungen der Befehle SET PATH und SET DEFAULT werden ignoriert.

Beispiele

. Das Beispiel zeigt die verschiedenen Ergebnisse von CURDIR():

```
? CURDIR("D":)     // Resultat: "", da Root-Verzeichnis
? CURDIR("C")      // Resultat: "CLIPPER5\SOURCE"
? CURDIR(C:)       // Resultat: "CLIPPER5\SOURCE"
? CURDIR()         // Resultat: "CLIPPER5\SOURCE"
? CURDIR("A")      // Resultat: "", Laufwerk nicht bereit
```

. Das folgende Beispiel wechselt das aktuelle DOS-Verzeichnis, wenn es nicht einer bestimmten Einstellung entspricht:

```
IF CURDIR("C:") <> "CLIPPER5\SOURCE"
  RUN CD \CLIPPER5\SOURCE
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB, Quelldatei ist EXAMPLEA.ASM

See Also:

DATE() Ermittelt das Systemdatum

Syntax

DATE() -> dSystem

R.ckgabe

DATE() liefert das Systemdatum als Datumswert zur.ck.

Anwendung

DATE() ist eine Datumsfunktion, die unter anderem folgende M.glichkeiten bietet: Speichervariablen mit dem Systemdatum zu initialisieren, Datumsvergleiche mit dem Systemdatum und Datumsberechnungen relativ zum Systemdatum durchzuf.hren.

Das Anzeigeformat f.r Datum h.ngt von der Einstellung .ber SET DATE ab. Die Voreinstellung ist mm/tt/jj.

Beispiele

Das folgende Beispiel zeigt DATE() mit verschiedenen Resultaten:

```
? DATE()                // Resultat: 13.09.90
? DATE() + 31           // Resultat: 13.10.90
? DATE() - 30           // Resultat: 13.08.90
dDatum := DATE()
? CMONTH(dDatum)        // Resultat: September
SET CENTURY ON
SET DATE ANSI
? DATE()                // Resultat: 1990.09.13
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

DAY() Ermittelt den Tag eines Monats

Syntax

DAY(<dDatum>) -> nTagImMonat

Parameter

<dDatum> ist das Datum, zu dem der Tag im Monat ermittelt werden soll.

R.ckgabe

DAY() gibt den Tag des Monats als numerischen Wert zwischen 1 und 31 zur.ck. Ist <dDatum> im Februar, werden Schaltjahre ber.cksichtigt. Ist <dDatum> ein leeres Datum, ist der R.ckgabewert 0.

Anwendung

DAY() ist eine Datumskonvertierungsfunktion und wird dazu benutzt, den Tag des Monats zu ermitteln. DAY() kann in Verbindung mit CMONTH() und YEAR() zur Formatierung von Datumsangaben verwendet werden. Weiter findet DAY() Anwendung bei Berechnungen mit Datumswerten.

Beispiele

. Das folgende Beispiel zeigt den unterschiedlichen Gebrauch der Funktion DAY():

```
? DATE()              // Resultat: 13.09.90
? DAY(DATE())         // Resultat: 13
? DAY(DATE()) + 1     // Resultat: 14
? DAY(CTOD(""))       // Resultat: 0
```

. Dieses Beispiel benutzt DAY() in Kombination mit CMONTH() und YEAR(), um ein Ausgabeformat zu bestimmen:

```
bBriefdatum := { x CMONTH(x) + " den " STR(DAY(x),2) +;
  ". "+STR(YEAR(x),4)
? EVAL(bBriefdatum, DATE())       // Resultat: September den 13. 1990
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

DBCREATE() Legt eine Datenbasis mit Strukturinformationen aus einem Array an

---

Syntax

DBCREATE(<cDatenbasis>, <aStruktur>) -> NIL

Parameter

<cDatenbasis> ist der Name der neu zu erstellenden Datenbasis als Zeichenkette. Optional k.nnen Laufwerkskennung und Verzeichnis mit angegeben werden. Wird keine Dateinamenserweiterung angegeben, so wird .DBF vorausgesetzt.

<aStruktur> ist ein Array, welches Strukturinformationen f.r die zu erstellende Datenbasis enth.lt. Die Struktur wird als eine Reihe von Subarrays angeben, wobei jedes Subarray ein Datenfeld beschreibt. Jedes Subarray besitzt dabei folgende Struktur:

Tabelle 5-7: DBCREATE() Felddefinitionen

Posititon	Metasymbol	DBStruct.ch
1	<cName>	DBS_NAME
2	<cTyp>	DBF_TYPE
3	<nL.nge>	DBF_LEN
4	<nDezimalstellen>	DBF_DEC

R.ckgabe

DBCREATE besitzt keinen R.ckgabewert

Anwendung

DBCREATE() ist eine Datenbankfunktion, welche eine Datenbasis erzeugt und die dazu n.tigen Strukturinformationen aus einem Array entnimmt. Das Array kann durch Programmierung oder durch die Funktion DBSTRUCT() angelegt werden. DBCREATE() ist vergleich- bar mit dem Befehl CREATE FROM, der Strukturinformationen aus einer Strukturdatei entnimmt. Die Strukturdatei kann .ber die Befehle CREATE oder COPY STRUCTURE EXTENDED angelegt sein.

Bevor DBCREATE() benutzt wird, mu. das Array <aStruktur> angelegt und mit Felddefinitionen zur Beschreibung der Dateistruktur gef.llt sein. Wenn die Subarrays mit Felddefinitionen versehen werden, sind einige Punkte zu beachten:

. Alle Feldattribute m.ssen mit dem jeweils f.r das Attribut selbst geltenden Datentyp angegeben werden. Das Dezimal-Attribut mu. angegben sein, auch wenn kein numerisches Feld definiert wird. In diesen F.llen ist eine 0 einzusetzen.

. Das Feldattribut Typ mu. mindestens mit dem ersten Buchstaben des jeweiligen Typs angegeben werden. L.ngere Angaben dienen nur der besseren Lesbarkeit. Zum Beispiel kann f.r ein Zeichenfeld sowohl "C" als auch "Character" angegeben sein.

. In Clipper k.nnen Zeichenfelder bis zu 65.520 Zeichen lang sein. Ungleich dem Befehl CREATE FROM mu. in DBCREATE() der h.herwertige Teil der L.nge nicht im Dezimal-Attribut angegeben werden. Die Feldl.nge kann direkt angegeben werden.

Um einen Bezug auf die verschiedenen Elemente des Felddefinitions-Subarrays zu erhalten, die lesbarer sind, enth.lt die Header-Datei mit dem Namen DBSTRUCT.CH #define-Anweisungen. Diese k.nnen dazu benutzt werden, um mit einem Namen auf das entsprechende Element des Subarrays zuzugreifen. Die Datei finden Sie im Verzeichnis CLIPPER5\INCLUDE.

Beispiele

. Das folgende Beispiel erzeugt ein leeres Array und f.gt mit der Funktion AADD() Felddefinitionen hinzu, bevor die Datenbasis angelegt wird:

```
aDbf := {}
AADD(aDbf,{ "Name", "C", 30, 0 })
AADD(aDbf,{ "Geburtstag", "D", 8, 0 })
AADD(aDbf,{ "Feminin", "L", 1, 0 })
AADD(aDbf,{ "Kredit", "N", 11, 2 })
AADD(aDbf,{ "Notizen", "M", 10, 0 })
//
DBCREATE("Personen", aDbf)
```

. Dieses Beispiel zeigt das Anlegen einer Dateistruktur, indem ein zweidimensionales Array deklariert wird, und verwendet Definitionen aus DBSTUCT.CH, um Elemente der Arraystruktur zu adressieren:

```
#include "DBStruct.ch"
//
LOCAL aDbf[1][4]
aDbf[1][ DBS_NAME ] := "Name"
aDbf[1][ DBS_TYPE ] := "Character"
aDbf[1][ DBS_LEN  ] := 30
aDbf[1][ DBS_DEC  ] := 0
//
DBCREATE( "Namen", aDbf )
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist DBStruct.ch

See Also:

DBEDIT()* Datens.tze in Tabellenform browsen

Syntax

```
DBEDIT([<nOben>], [<nLinks>], [<nUnten>], [<nRechts>],
       [<acSpalten>],
       [<cAnwenderFunktion>],
       [<acAnzeigeMaske> |<cAnzeigeMaske>],
       [<acSpaltenKopf> |<cSpaltenKopf>],
       [<acKopfTrenner> |<cKopfTrenner>],
       [<acSpaltenTrenner> |<cSpaltenTrenner>],
       [<acFussTrenner> |<cFussTrenner>],
       [<acSpaltenfuss> |<cSpaltenfuss>]) -> NIL
```

Parameter

<nOben>, <nLinks>, <nUnten>, <nRechts> sind Fensterkoordinaten und geben den Bereich des DBEDIT()-Fensters an. Bildschirmzeilen k.nnen von 0 bis MAXROW() und Bildschirmspalten von 0 bis MAXCOL() angegeben werden. Werden Parameter weggelassen, werden Standardwerte angenommen: 0, 0, MAXROW(), MAXCOL().

<acSpalten> ist ein Array mit Zeichenkettenausr.cken, welche die Feldnamen der Datenbasis oder Ausdr.cke enthalten k.nnen, deren Werte in den einzelnen Spalten dargestellt werden sollen. Wird kein Parameter angegeben, so werden die Spalten mit den Feldern der Datenbasis im aktuellen Arbeitsbereich gef.llt.

<cAnwenderFunktion> ist der Name einer anwenderdefinierten Funktion, die immer dann ausgef.hrt wird, wenn eine undefinierte Taste gedr.ckt wird oder keine Tastencodes im Tastaturpuffer enthalten sind. Der Funktionsname wird als Zeichenkette ohne folgende Klammern und Parameter angegeben. Beachten Sie, da. das Verhalten von DBEDIT sich .ndert, sobald dieser Parameter weggelassen wird. Weitere Informationen folgen sp.ter.

<acAnzeigeMaske> ist ein paralleles Array von <cAnzeigeMaske> und dazu bestimmt, die Anzeige der Spalten zu formatieren. Wenn anstelle des Arrays eine Zeichenfolge <cAnzeigeMaske> angegeben wird, erhalten alle Spalten das gleiche Format. Siehe auch TRANSFORM() oder @...SAY...PICTURE, um n.here Informationen .ber das Anzeigeformat zu erhalten.

<acSpaltenKopf> ist ein paralleles Array mit Zeichen, die als Spalten.berschriften verwendet werden. Wird <cSpaltenkopf> angegeben, so erhalten alle Spalten die gleiche .berschrift. Um eine mehrzeilige .berschrift zu erzeugen, kann die Zeichenfolge ein Semikolon enthalten, an dessen Stelle die Zeile umgebrochen wird. Wird kein Parameter angegeben, werden die Spalten.berschriften aus <acSpalten> entnommen oder, wenn <acSpalten> ebenfalls fehlt, die Feldnamen der aktuellen Datenbasis eingesetzt.

<acKopfTrenner> ist ein paralleles Array mit Zeichen, welches die Trennung zwischen .berschrift und Feldbereich angibt. Wird <cKopfTrenner> anstelle des Arrays eingesetzt, so werden die .berschriften aller Spalten gleich aussehen. Ist kein Parameter angegeben, wird eine Doppellinie als Trennung eingesetzt.

<acSpaltenTrenner> ist ein paralleles Array mit Zeichen, welches die vertikale Trennung zwischen den Spalten definiert. Wird anstelle des Arrays <cSpaltenTrenner> eingesetzt, so erhalten alle Spalten die gleiche Trennung. Ist kein Parameter angegeben, wird eine einfache vertikale Linie eingesetzt.

<acFussTrenner> ist ein paralleles Array mit Zeichen, mit denen eine horizontale Trennung zwischen Feldbereich und Fu.zeile f.r jede Spalte erfolgt. Wird anstelle des Arrays <cFussTrenner> eingesetzt, so erhalten alle Spalten die gleiche Trennung. Ist kein Parameter angegeben, entf.llt eine Trennung zwischen Feldbereich und Fu.zeile.

<acSpaltenFuss> ist ein paralleles Array mit Zeichenketten, welche die

Fu.zeilen f.r jede Spalte beschreiben. Wird <cSpaltenFuss> angegeben, erhalten alle Spalten die gleichen Fu.zeilen. Um mehrzeilige Fu.zeilen zu erzeugen, kann die Zeichenkette ein Semikolon enthalten, an dessen Stelle die Zeile umgebrochen wird. Ist der Parameter nicht angegeben, wird keine Fu.zeile angezeigt.

R.ckgabe

DBEDIT() besitzt keinen R.ckgabewert.

Anwendung

DBEDIT() ist eine anwenderorientierte Funktion, welche Datens.tze aus einem oder mehreren Arbeitsbereichen in Tabellenform darstellt. Das DBEDIT()-Fenster ist ein Gitter von Zellen, aufgeteilt in Spalten und Zeilen. Spalten stehen f.r Datenfelder und Zeilen f.r Datens.tze. Jede Spalte ist durch den Parameter <acSpalten> definiert. Die Weite jeder Spalte wird bestimmt von der Angabe in <cSpalte>.

Alle Cursorbewegungstasten werden von DBEDIT() unterst.tzt, einschlie.lich der Tasten [Bild.], [Bild.], [Pos1], [Ende], den vier Pfeiltasten und alle g.ltigen [Strg]-Tastenkombinationen, die Cursorbewegungen steuern. Die folgende Tabelle zeigt die Bedientasten f.r DBEDIT(), wenn keine Anwenderfunktion angegeben ist:

Tabelle 5-8: DBEDIT() aktive Bedientasten

Taste	Inkey.ch	Aktion
PfeilAufw.rts	K_UP	Eine Zeile h.her
PfeilAbw.rts	K_DN	Eine Zeile tiefer
Linkspfeil	K_LEFT	Eine Spalte links
Rechtspfeil	K_RIGHT	Eine Spalte rechts
Strg+Linkspfeil	K_CTRLLEFT	Eine Spalte links rollen
Strg+Rechtspfeil	K_CTRLRIGHT	Eine Spalte rechts rollen
Pos1	K_HOME	Linke Bildschirmspalte
Ende	K_END Rechte	Bildschirmspalte
Strg+Pos1	K_CTRLHOME	Linke Tabellenspalte
Strg+Ende	K_CTRLEND	Rechte Tabellenspalte
Bild.	K_PGUP Ein	Bild h.her
Bild.	K_PGDN Ein	Bild tiefer
Strg+Bild.	K_CTRLPGUP	Erste Tabellenzeile
Strg+Bild.	K_CTRLPGDN	Letzte Tabellenspalte
Return	K_RETURN	Abbruch von DBEDIT()
Esc	K_ESC	Abbruch von DBEDIT()

Wenn der Parameter <cAnwenderFunktion> angegeben wird, sind alle Bedientasten weiterhin g.ltig, mit Ausnahme der Tasten [Esc] und [Return]. Wenn DBEDIT eine Anwenderfunktion aufruft, werden ihr automatisch zwei Argumente .bergeben:

. Der aktuelle Modus als numerischer Wert
. Ein Index auf die aktuelle Spalte in <acSpalte> als numerischer Wert

Das Modus-Argument zeigt den aktuellen Status von DBEDIT() an, nachdem das letzte Tastenkommando abgearbeitet ist und die Anwenderfunktion aufgerufen wird. Die folgende Tabelle beschreibt die Modi-Argumente:

Tabelle 5-9: DBEDIT() Moduswerte

Modus	Ufunc.ch	Beschreibung
0	DB_IDLE	Ruhezustand
1	DB_PASTTOP	Versuch BOF() zu .berschreiten
2	DB_PASTEND	Versuch EOF() zu .berschreiten
3	DB_EMPTY	Datenbasis enth.lt keine Datens.tze
4	DB_EXCEPT	Ung.ltiger Tastendruck

Das Index-Argument zeigt auf die Position der aktuellen Spalte in <acSpalten>. Wird der Parameter <acSpalten> weggelassen, zeigt dieses Argument auf die Feldposition der aktuellen Dateistruktur. Der Feldname kann dabei .ber die Funktion FIELD() ermittelt werden.

Wenn eine Anwenderfunktion aufgerufen wird, mu. sie nach Abarbeitung der

darin enthaltenen Anweisungen einen Wert zur.ckliefern, der DBEDIT mitteilt, wie als n.chstes zu verfahren ist. Die folgende Tabelle enth.lt die Beschreibung der verschiedenen R.ckgabewerte:

Tabelle 5-10: DBEDIT() Anwenderfunktion R.ckgabewerte

Wert	Ufunc.ch	Beschreibung
0	DB_QUIT	DBEDIT() verlassen
1	DB_CONT	DBEDIT() fortsetzten
2	DB_REFRESH	Daten neu einlesen, Bildaufbau erneuern und fortfahren mit DBEDIT()
3		APPEND-Modus (nicht empfohlen)

Die Anwenderfunktion wird in drei unterschiedlichen F.llen aufgerufen:

. Ung.ltiger Tastendruck: Dies ist der Fall, wenn DBEDIT() einen Tastencode aus dem Tastaturpuffer holt und dieser nicht von DBEDIT() bearbeitet werden kann. Einige noch ausstehende Tasten verbleiben so lange im Tastaturpuffer, bis sie entweder von einer Anwenderfunktion ben.tigt werden, oder BEDIT() einen neuen Tastencode anfordert.

. DBEDIT() geht in Ruhezustand: Alle Tastencodes sind abgearbeitet. Dies ist der Fall, wenn der Tastaturpuffer leer ist oder der Bildschirm neu aufgebaut wird. In diesem Zustand wird die Anwenderfunktion nur einmal aufgerufen.

. Dateigrenzen erreicht: Dies ist die gleiche Situation wie im Ruhezustand. Alle ausf.hrbaren Tasten sind abgearbeitet und es wird die Anwenderfunktion einmal aufgerufen, um eventuell eine Statusmeldung anzuzeigen.

Die Funktion DBEDIT() ist voll re-entrant. Das bedeutet, da. innerhalb der Anwenderfunktion nochmals DBEDIT() aufgerufen werden kann. Durch diese Verschachtelung von DBEDIT k.nnen mehrere Browse-Fenster zur gleichen Zeit am Bildschirm vorhanden sein.

DBEDIT ist eine Funktion, die aus Kompatibilit.tsgr.nden beibehalten wurde, allerdings f.r neue Applikationen nicht mehr empfohlen wird. Eine wesentlich elegantere Methode besteht in der Anwendung der TBROWSE-Objektklassen. Lesen Sie dazu das Kapitel Objektklassen des Referenz-Handbuches 2.

Beispiele

Dies Beispiel demonstriert den Einsatz von DBEDIT() unter Angabe einer Anwenderfunktion:

```
USE Kunden INDEX KundenNr NEW
USE Verkauf INDEX Produkt NEW
SET RELATION TO KdNr INTO Kunden
//
acSpalten :=  {"Produkt","Menge","Kunden->Kunde"}
DBEDIT( 4, 0, 22, 79, acSpalten, "MyUDF" )
```

Dateien

Bibliothek ist EXTEND.LIB, Header-Datei sind Ufunc.ch und Key.ch

---

See Also:

DBEVAL() Wertet einen Code-Block f.r jeden .ber Bereich und Bedingung eingrenzbaren Datensatz aus

---

Syntax

```
DBEVAL(<bBlock>,
      [<bForBedingung>],
      [<bWhileBedingung>],
      [<nNextBedingung>],
      [<nDatensatz>],
      [<lRest>]) -> NIL
```

Parameter

<bBlock> ist der Code-Block, der f.r jeden Datensatz ausgef.hrt wird.

<bForBedingung> ist eine optionale Bedingung, spezifiziert als Code-Block, welche f.r jeden Datensatz eines festgelegten Bereiches ausgef.hrt wird. Sie liefert die gleiche Eigenschaft wie die FOR-Bedingung eines Datensatzverarbeitungs-Befehls (LIST ... FOR).

<bWhileBedingung> ist eine optionale Bedingung, spezifiziert als Code-Block, welche ab dem aktuellen Datensatz durchgef.hrt wird, bis die Bedingung false (.F.) liefert. Sie liefert damit die gleiche Eigenschaft wie die WHILE-Bedingung eines Datensatzverarbeitungs-Befehls (LIST...WHILE).

<nNextBedingung> ist ein numerischer Wert, der die Anzahl der zu verarbeitenden Datens.tze angibt, beginnend beim aktuellen Satzzeiger (RECNO()). Sie entspricht damit einer NEXT-Bedingung in einem Datensatzverarbeitungs-Befehl.

<nDatensatz> ist ein numerischer Wert, der die Datensatznummer angibt, f.r welche <bBlock> durchgef.hrt werden soll. Dieser Parameter entspricht damit der RECORD-Bedingung.

<lRest> ist ein logischer Wert, der angibt, ob DBEVAL() f.r alle Datens.tze, oder vom aktuellen Datensatz bis zum Ende der Datei durchgef.hrt wird. Dies entspricht den REST- und ALL-Angaben in einem Datenbank-Befehl. Lautet der Wert true (.T.), ist der Bereich REST, ansonsten gilt ALL. Wird der Parameter nicht angegeben, so gilt die Einstellung ALL.

R.ckgabe

DBEVAL besitzt keinen R.ckgabewert.

Anwendung

DBEVAL() ist eine Datenbankfunktion, welche einen einzelnen Code-Block f.r jeden Datensatz des aktuellen Arbeitsbereiches ausf.hrt, der den angegebenen Bedingungen entspricht. F.r jeden Datensatz, welcher den Bereichsangaben und den Bedingungen entspricht, wird der Proze. durchgef.hrt.

Als Voreinstellung verwendet DBEVAL() die Datenbasis im aktuellen Arbeitsbereich. Die Funktion kann auch auf nichtselektierte Arbeitsbereiche angewendet werden, indem man den Alias-Bezeicher vor den Funktionsnamen stellt.

DBEVAL() ist zu vergleichen mit der Funktion AEVAL(), welche einen Code-Block f.r jedes Arrayelement auswertet. .hnlich wie AEVAL(), kann DBEVAL() als einfaches Mittel zur Erstellung von anwenderdefinierten Befehlen (User Defined Commands) verwendet werden. Viele Standard-Befehle von Clipper 5.0 sind tats.chlich auf der Verwendung von DBEVAL() aufgebaut.

Lesen Sie den Abschnitt Code-Bl.cke im Kapitel Grundkonzepte, um n.here Informationen .ber die Theorie und Syntax von Code-Bl.cken zu erhalten. Weitere Informationen .ber Bereichsangaben und Bedingungen der Datensatzauswahl erhalten Sie im Abschnitt Datenbanksystem des Kapitels

Grundkonzepte. Weitere Informationen k.nnen ebenfalls aus der Clipper 5.0 Header-Datei STD.CH entnommen werden, die eine Reihe von Datenbankbefehlen enth.lt, welche den Einsatz der Funktion DBEVAL() demonstrieren.

Beispiele

Das folgende Beispiel verwendet DBEVAL(), um die COUNT()-Funktion zu implementieren. COUNT() ist eine Anwenderfunktion, die eine Anzahl von Datens.tzen in einem definierten Bereich .ber eine definierte Bedingung z.hlt. Der Bereich wird .ber ein Array an COUNT() .bergeben. Dies Beispiel ist insofern sehr interessant, da es zeigt, wie ein anwenderdefinierter Befehl (UDC) definiert wird, um ein Array zu erzeugen, welches die Bereichs- und Bedingungsangaben entgegen nehmen kann:

```
#command CREATE SCOPE <aScope> [FOR <for>] [WHILE <while>];
  [NEXT <next>] [RECORD <rec>] [<rest:REST>] [ALL];
 => <aScope> := {<{for}>, <{while}>, <next>, <rec>, <.rest.>}
//
#define FOR_COND    1
#define WHILE_COND  2
#define NEXT_SCOPE  3
#define REC_SCOPE   4
#define REST_SCOPE  5

LOCAL MySet, MyCount
USE Kunden NEW
CREATE SCOPE MySet FOR Kunden = "Schmitz" WHILE Ort = "Berlin"
MyCount := COUNT(MySet)
RETURN

FUNCTION COUNT(aScope)
LOCAL nCount := 0
DBEVAL({ nCount++}, aScope[ FOR_COND ],;
  aScope[ WHILE_COND ], aScope[ NEXT_SCOPE ],;
  aScope[ REC_SCOPE ],    aScope[ REST_SCOPE ] )
RETURN nCount
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

DBF()* Ermittelt den Aliasbezeichner des aktuellen Arbeitsbereiches

---

Syntax

```
DBF() -> cAlias
```

R.ckgabe

Liefert den Aliasnamen des aktuellen Arbeitsbereiches als Zeichenkette. Ist im aktuellen Arbeitsbereich keine Datei ge.ffnet, liefert DBF() eine leere Zeichenkette ("").

Anwendung

DBF() ist nur aus Gr.nden der Kompatibilit.t implementiert. Unter Clipper kann diese Funktion auch durch ALIAS() ohne Angabe eines Parameters realisiert werden.

Da DBF() eine Kompatibilit.tsfunktion ist, wird sie f.r neue Applikationen nicht empfohlen. Benutzen Sie die leistungsf.higere ALIAS()-Funktion.

Dateien

Bibliothek ist EXTEND.LIB

See Also:

DBFILTER() Ermittelt die aktuelle Filterbedingung

Syntax

DBFILTER() -> cFilter

R.ckgabe

DBFILTER() liefert die Filterbedingung des aktuellen Arbeitsbereiches in einer Zeichenfolge. Ist kein Filter definiert, liefert DBFILTER() einen Nullstring ("").

Anwendung

DBFILTER() ist eine Datenbankfunktion und wird benutzt, um aktuelle Filterbedingungen zu sichern, um diese sp.ter .ber den Makro-Operator (&) wieder herstellen zu k.nnen.

Diese Funktion arbeitet auf einfache Weise mit DBRELATION(), RECNO() und DBRSELECT() zusammen, um den kompletten Status eines Arbeitsbereiches tempor.r zu sichern.

Zu jedem Arbeitsbereich kann ein eigener Filter gesetzt sein. Um Filterkriterien aus einem anderen Arbeitsbereich als dem aktiven zu ermitteln, mu. dem Funktionsnamen ein Aliasbezeichner vorangestellt werden.

Hinweise

. Deklarierte Variablen: Treten in den Filterbedingungen Referenzen auf lokale oder statische Variablen auf, ist es m.glich, da. DBFILTER() undefinierbare Ergebnisse liefert, wenn die Funktion in einem Makro zur Laufzeit eines Programms ausgef.hrt wird.

Beispiele

. Das Beispiel .ffnet zwei Datenbanken, setzt zwei Filter und zeigt diese dann an:

```
USE Kunden INDEX Kunden NEW
SET FILTER TO Letzter = "Schmidt"
USE Inventar INDEX Inventar NEW
SET FILTER TO KundenNr = "Smi001"
SELECT kunden
//
? DBFILTER()                  // Resultat: "Letzter = Schmidt"
? Inventar->(DBFILTER())      // Resultat: "KundenNr = Smi001"
```

. Die folgende Anwenderfunktion SaveQry() benutzt DBFILTER() dazu, um die aktuelle Filterbedingung abzuspeichern. Die Speicherdatei erh.lt als Dateinamenserweiterung ".QWY, um anzuzeigen, da. sie eine Filterbedingung enth.lt:

```
FUNCTION SaveQry(cQryName)
LOCAL cFilter := DBFILTER()
  SAVE ALL LIKE cFilter TO (cQryName + ".QWY")
RETURN NIL
```

. Sp.ter kann auf diesen Filter wieder zugegriffen werden, wenn die Anwenderfunktion SetFilter() angewandt wird. Diese Funktion greift auf die Query-Datei zu und setzt den abgespeicherten Filter neu.

```
FUNCTION SetFilter(cQryName)
  RESTORE FROM (cQryName+".QWY") ADDITIVE
  SET FILTER TO &cFilter
RETURN NIL
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

DBRELATION() Ermittelt den Verbindungsausdruck einer spezifizierten Relation

---

Syntax

DBRELATION(<nRelation>) -> cVerbindungsAusdruck

Parameter

<nRelation> ist die Position der Relation in der Liste der aktuellen Relationen eines Arbeitsbereiches. Die Relationen existieren in der Reihenfolge, wie sie .ber den Befehl SET RELATION definiert wurden.

R.ckgabe

DBRELATION() liefert den Verbindungsausdruck der in <nRelation> angegebenen Verbindung als Zeichenkette zur.ck. Existiert diese Relation nicht, wird eine leere Zeichenfolge zur.ckgegeben.

Anwendung

DBRELATION() ist eine Datenbankfunktion und wird in Kombination mit DBRSELECT() dazu benutzt, um eine Verkn.pfung und den Arbeitsbereich, welche mit SET RELATION bestimmt wurden, abzufragen. Relationen sind eine Folge von Verbindungsdefinitionen zu anderen Arbeitsbereichen, die in der gleichen Folge .ber SET RELATION gesetzt wurden.

DBRELATION() liefert den Verkn.pfungsausdruck, welcher durch TO bestimmt wurde. DBRSELECT liefert den Arbeitsbereich, welcher durch INTO bestimmt wurde.

Da jeder Arbeitsbereich seine eigene Relation haben kann, ist es m.glich, mit DBRELATION() die Relation jedes Arbeitsbereiches zu ermitteln, indem Sie den Aliasbezeichner voranstellen. Siehe dazu auch das Beispiel weiter unten.

Hinweise

Deklarierte Variablen: Treten in den Relationen Referenzen auf lokale oder statische Variablen auf, ist es m.glich, da. DBRELATION() undefinierbare Ergebnisse liefert, wenn die Funktion in einem Makro zur Laufzeit eines Programms ausgef.hrt wird.

Beispiele

. Das folgende Beispiel .ffnet drei Datenbanken, setzt zwei Relationen und zeigt die zweite Relation an:

```
USE   Rechnung INDEX Rechnung NEW
USE   Bestell INDEX Bestell NEW
USE   Kunden INDEX Kunden NEW
SET   RELATION TO KundenNr INTO Rechnung;
      TO BestellNr INTO Bestell
//
? DBRELATION(2)              // Resultat: BestellNr
```

. Sp.ter k.nnen Sie dann auf den Ausdruck der Relation auch aus anderen Arbeitsbereichen zugreifen:

```
USE Archiv NEW
? Kunden->(DBRELATION(2))   // Resultat: BestellNr
```

. Das letzte Beispiel zeigt die Verwendung von DBRELATION() und DBRSELECT(), die beide Ergebnisse in einem Array mit zwei Elementen zur.ckgeben:

```
FUNCTION GetRelation(nRelation)
RETURN {DBRELATION(nRelation), DBRSELECT(nRelation)}
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

DBRSELECT() Ermittelt den Arbeitsbereich einer Relation

Syntax

DBRSELECT(<nRelation>) -> nArbeitsbereich

Parameter

<nRelation> ist die Position in der Liste der aufgebauten Relationen.

R.ckgabe

DBRSELECT() liefert die Nummer des Arbeitsbereiches einer durch <nRelation> spezifizierten Relation als ganzzahligen, numerischen Wert. Ist f.r <nRelation> keine Relation aufgebaut, wird Null zur.ckgegeben.

Anwendung

DBRSELECT() ist eine Datenbankfunktion und wird in Kombination mit DBRRELATION() dazu benutzt, um eine Verkn.pfung und den Arbeitsbereich, welche mit SET RELATION bestimmt wurden, abzufragen. Relationen sind eine Folge von Verbindungsdefinitionen zu anderen Arbeitsbereichen, die in der gleichen Folge .ber SET RELATION gesetzt wurden. DBRSELECT liefert den Arbeitsbereich, welcher durch INTO bestimmt wurde. DBRELATION() liefert den Verkn.pfungsausdruck, welcher durch TO bestimmt wurde. Um anstelle der Bereichsnummer den entsprechenden Aliasbezeicher zu erhalten, kann der Ausdruck ALIAS(DBRSELECT(<nRelation>)), benutzt werden.

Als Voreinstellung arbeitet DBRSELECT() im aktuellen Arbeitsbereich. Die Funktion kann durch Voranstellen des Aliasbezeichners auch auf andere Arbeitsbereiche angewandt werden.

Beispiele

. Das folgende Beispiel .ffnet drei Datenbanken und setzt zwei Relationen aus dem aktuellen Arbeitsbereich. Danach wird der zweite Verbindungsausdruck angezeigt:

```
USE   Rechnung INDEX Rechnung NEW
USE   Bestell INDEX Bestell NEW
USE   Kunden INDEX Kunden NEW
SET   RELATION TO KundenNr INTO Rechnung;
  TO BestellNr INTO Bestell
//
? DBRELATION(2), DBRSELECT(2)        // Resultat: BestellNr 2
? ALIAS(DBRSELECT(2))                // Resultat: Bestell
```

. Sp.ter k.nnen Sie auf diese Informationen auch zugreifen, wenn Sie sich nicht in diesem Arbeitsbereich befinden:

```
USE archiv NEW
? Kunden->(DBRELATION(2), DBRSELECT(2)) // Resultat: BestellNr 2
? Kunden->(ALIAS(DBRSELECT(2))).// Resultat: Bestell
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

DBSTRUCT() Erzeugt ein Array mit den Strukturinformationen einer Datenbank

---

Syntax

DBSTRUCT() -> aStruktur

R.ckgabe

DBSTUCT() liefert die Struktur einer Datenbank in Form eines Arrays, das genausoviele Elemente besitzt, wie Felder in der Datenbank sind. Jedes Element ist ein Subarray mit den Felddefinitionen eines jeden Feldes. Das Subarray besitzt folgendes Format:

Tabelle 5-11: DBSTRUCT() R.ckgabe Arrays

Posititon	Metasymbol	DBStruct.ch
1	<cName>	DBS_NAME
2	<cType>	DBS_TYPE
3	<nL.nge>	DBS_LEN
4	<nDezimalstellen>	DBS_DEC

Anwendung

DBSTRUCT() ist eine Datenbankfunktion, die .hnlich arbeitet wie COPY STRUCTURE EXTENDED, indem sie ein Array mit Strukturinformationen erzeugt, anstatt diese Informationen in einer separaten Datei abzulegen. Eine weitere Funktion DBCREATE() kann aus einem .ber DBSTRUCT() erzeugten Array wieder eine Datenbasis erzeugen. Als Voreinstellung arbeitet DBSTRUCT() mit der Datenbank im aktuellen Verzeichnis. Die Funktion kann auch auf Datenbanken in anderen Arbeitsbereichen angewendet werden, indem ein Aliasbezeichner vorangestellt wird.

Beispiele

Das folgende Beispiel .ffnet zwei Datenbanken und erzeugt ein Array mit Strukturinformationen durch DBSTRUCT() mit einem Aliasbezeichner. Die Feldnamen werden .ber AEVAL() aufgelistet:

```
#include "DBStruct.ch"
LOCAL aStruktur
USE Kunden NEW
USE Rechnung NEW
//
aStruktur := Kunden->(DBSTRUCT())
AEVAL(aStruct, { | aField |  QOUT(aField[DBS_NAME])} )
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist DBStruct.ch

See Also:

DELETED() Liefert den L.sch-Status des aktuellen Datensatzes

Syntax

DELETED() -> lGel.scht

R.ckgabe

DELETED() liefert den Wert true (.T.), wenn der aktuelle Datensatz zum L.schen markiert ist, andernfalls false (.F.). Ist keine Datenbank ge.ffnet, wird false (.F.) zur.ckgegeben.

Anwendung

DELETED() ist eine Datenbankfunktion, die feststellt, ob ein Datensatz durch die Anweisung DELETE zum L.schen markiert worden ist. Wenn in verschiedenen Arbeitsbereichen Datenbanken ge.ffnet sind, steht jeder Arbeitsbereich auf einem aktuellen Datensatz, wobei jeder Satz eine eigene L.schmarkierung haben kann.

Als Voreinstellung arbeitet DELETED() auf dem Datensatz im aktuellen Arbeitsbereich. Die Funktion kann auf andere Arbeitsbereiche durch Voranstellen eines Aliasbezeichners angewendet werden.

In Applikationen wird DELETED() haupts.chlich dazu ben.tigt, um Abl.ufe von Prozessen zu steuern, oder diese als Status am Bildschirm anzuzeigen.

Beispiele

. Dieses Beispiel zeigt die Anwendung von DELETED() auf den aktuellen und auf einen zweiten Arbeitsbereich:

```
USE Kunden NEW
USE Verkauf NEW
//
? DELETED()              // Resultat: .F.
DELETE
? DELETED()              // Resultat: .T.
? Kunden->(DELETED())    // Resultat: .F.
```

. Das n.chste Beispiel zeigt, wie DELETED() f.r Statusanzeigen auf dem Bildschirm oder in Reports angezeigt werden kann:

```
@ 1, 65 SAY IF(DELETED(), "Inaktiv", "Aktiv  ")
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

DESCEND() Erzeugt einen Indexschl.ssel f.r absteigende Sortierung

---

Syntax

DESCEND(<exp>) -> expKomplement

Parameter

<exp> ist ein g.ltiger Ausdruck vom Typ Zeichen, Datum, Logisch oder Numerisch. Datenbankfelder vom Typ Memo werden wie Zeichen behandelt.

R.ckgabe

DESCEND() liefert einen Wert vom gleichen Typ wie <exp> in komplement.rer Form.
Eine Ausnahme bilden Datumswerte, sie werden als numerische Werte zur.ckgegeben.
Ist CHR(0) als Parameter angegeben, wird auch CHR(0) zur.ckgegeben.

Anwendung

DESCEND() ist eine Konvertierungsfunktion, die einen Komplement.rwert von <exp> erzeugt, wodurch es erm.glicht wird, Indexausdr.cke in absteigender Reihenfolge zu erzeugen. Dazu geben Sie den zu indizierenden Wert als Argument an die Funktion DESCEND() weiter. Wird nach einem solchen Indexausdruck sp.ter mit SEEK gesucht, so ist ebenfalls der Suchausdruck mit der Funktion DESCEND() zu konvertieren.

Beispiele

. Dieses Beispiel benutzt die Funktion DESCEND(), um nach dem Datum in absteigender Folge zu indizieren:

```
USE Verkauf NEW
INDEX ON DESCEND(BestellDat) TO BESTELNG
```

. Sp.ter kann DECEND() benutzt werden, um mit SEEK den gew.nschten Audruck zu suchen:

```
SEEK DESCEND(dSuch)
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

DEVOUT() Schreibt einen Wert auf die aktuelle Ausgabeeinheit

Syntax

DEVOUT(<exp>) -> NIL

Parameter

<exp> ist ein anzuzeigender Wert.

R.ckgabe

DEVOUT() besitzt keinen R.ckgabewert.

Anwendung

DEVOUT ist eine Ausgabefunktion, die einen Wert auf die mit SET DEVICE TO angegebene Ausgabeeinheit schreibt. Die Ausgabe beginnt dabei an der aktuellen Cursor- oder Druck-Position. In der Header-Datei STD.CH wird diese Funktion zur Realisierung des @...SAY Befehls verwendet.

Beispiele

Die folgenden zwei Beispiele zeigen den Einsatz der Funktion DEVOUT() auf zwei unterschiedlichen Wegen, aber mit dem gleichen Resultat:

```
@ 10,10 SAY "Hallo"

// Wird durch die Haeder-Datei wie folgt umgesetzt und kann auch wie folgt
// angewendet werden:

DEVPOS(10,10)
DEVOUT("Hallo")
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

DEVPOS() Bewegt den Cursor oder Druckkopf an eine neue Position

---

Syntax

DEVPOS(<nZeile>, <nSpalte>) -> NIL

Parameter

<nZeile> und <nSpalte> bestimmen die neue Position des Cursors oder des Druckkopfes.

R.ckgabe

DEVPOS besitzt keinen R.ckgabewert.

Anwendung

DEVPOS() ist eine Umgebungsfunktion, welche den Cursor oder den Druckkopf in eine definierte Position bewegt.

Ist SET DEVICE auf SCREEN gesetzt, bewegt die Funktion DEVPOS() den Bildschirmcursor auf die angegebene Position und die Werte f.r ROW() und COL() werden aktualisiert.

Ist SET DEVICE auf PRINT gesetzt, wird der Druckkopf positioniert und die Werte f.r PROW() und PCOL() aktualisiert. Der Druckkopf wird dabei durch eine bestimmte Anzahl von Zeilen- und Seitenvorsch.ben (LF bzw FF) auf die entsprechende Zeile positioniert.
Ist die aktuelle Einstellung von SET MARGIN TO gr..er Null, dann wird sie zur Spaltenposition <nSpalte> hinzuaddiert. Der Druckkopf wird an die entsprechende Position gebracht, wobei PCOL() und PROW() aktualisiert werden. Sind desweiteren <nZeile> und <nSpalte> kleiner als die aktuelle Einstellung von PROW() und PCOL(), wird der Druckkopf nach folgenden Regeln bewegt:

. Ist <nZeile> kleiner als PROW(), wird automatisch ein EJECT (CHR(12)) an den Drucker gesendet, gefolgt von so vielen Zeilenvorsch.ben (CHR(10)), bis die Position <nZeile> erreicht ist.

. Ist <nSpalte> inclusive des aktuellen SET MARGIN-Wertes kleiner als PCOL(), so wird ein Carriage Return-Zeichen (CHR(13)), gefolgt von einer Anzahl von Leerzeichen gesendet, bis <nSpalte> erreicht wird.

Um diese Regeln zu umgehen, kann SETPRC() zum Zur.cksetzen von PROW() und PCOL() eingesetzt werden, bevor DEVICEPOS() angewendet wird. Weitere Informationen entnehmen Sie bitte der Beschreibung der Funktion SETPRC() in diesem Handbuch.

Ist die Druckausgabe mit dem Befehl SET PRINTER TO in eine Datei umgelenkt worden, so wird anstatt des Druckkopfes diese Datei aktualisiert.

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

DIRECTORY() Erzeugt ein Array mit Datei- und Verzeichnisinformationen

---

Syntax

DIRECTORY([<cDateimaske>], [<cAttribute>]) -> aVerzeichnis

Parameter

<cDateimaske> spezifiziert das Laufwerk, das Verzeichnis und eine Dateimaske, um nach Eintr.gen zu suchen. Platzhalter nach DOS-Standardkonvention sind erlaubt. Wird dieser Parameter nicht angegeben, wird die Standardeinstellung *.* angenommen.

<cAttribute> gibt an, da. Dateien mit speziellen Dateiattributen bei der Verzeichnissuche eingeschlossen werden. <cAttribute> kann ein oder mehrere der folgenden Zeichen enthalten:

Tabelle 5-12: DIRECTORY() Dateiattribute

Attribut	Erl.uterung
H	Schlie.t versteckte (Hidden) Dateien ein
S	Schlie.t Systemdateien ein
D	Schlie.t Verzeichnisnamen (Directories) ein
V	Ermittelt den Datentr.gernamen (VOLUME)

Normale Dateien werden bei der Verzeichnissuche immer ber.cksichtigt, es sei denn, das Attribut V wurde angegeben.

R.ckgabe

ADIR() liefert ein Array mit Subarrays, wobei jedes Subarray Informationen .ber jeden mit <cDateimaske> gefundenen Verzeichniseintrag enth.lt. Die Subarrays besitzen folgende Struktur:

Tabelle 5-13: DIRECTORY() Subarray-Struktur

Position	Metasymbol	Directry.ch
1	<cName>	F_NAME
2	<cGr..e>	F_SIZE
3	<dDatum>	F_DATE
4	<cUhrzeit>	F_TIME
5	<cAttribut>	F_ATT

Sind keine Dateien gefunden, oder ist eine ung.ltige <Dateimaske> angegeben worden, liefert DIRECTORY() ein leeres Array ({}) zur.ck.

Anwendung

DIRECTORY() ist eine Environment-Funktion, welche Informationen .ber Dateien im aktuellen oder angegebenen Verzeichnis liefert. Sie ist .hnlich wie ADIR(), jedoch liefert sie ein einziges Array, anstatt die Ergebnisse in einer Reihe bereits existierender Arrays abzulegen, die per Referenz .bergeben werden.

DIRECTORY() kann benutzt werden, um Aktionen mit einer Gruppe von Dateien durchzuf.hren. In Kombination mit der Funktion AEVAL() kann ein Code-Block definiert werden, welcher Dateien benutzt, die mit <cDateimaske> ausgew.hlt wurden.

Um die verschiedenen Elemente eines jeden Subarrays leichter zu identifizieren, ist eine Header-Datei DIRECTRY.CH vorhanden, die Definitionen der Subarrays beschreibt. Diese Datei ist im Verzeichnis \CLIPPER\INCLUDE enthalten.

Beispiele

Das folgende Beispiel erzeugt ein Array mit Informationen des aktuellen Verzeichnisses und listet die Namen der gefundenen Dateien mit AEVAL() und QOUT():

```
#include "DIRECTRY.CH"

aDateien := DIRECTORY( "*.*", "D" )
AEVAL( aDateien, { | aFile QOUT(aFile[F_NAME]) } )
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

DISKSPACE() Liefert den freien Speicherplatz eines Laufwerks

Syntax

DISKSPACE([<nLaufwerk>]) -> nByteFrei

Parameter

<nLaufwerk> ist die Nummer des Laufwerks, f.r das der freie Speicherplatz ermittelt werden soll. Dabei steht 1 f.r A:, 2 f.r B:, 3 f.r C:, usw. Wird <nLaufwerk> weggelassen oder 0 angegeben, wird der Speicherplatz f.r das aktuelle Laufwerk ermittelt.

R.ckgabe

Es wird die Gr..e des freien Speicherplatzes in Byte als numerischer Wert zur.ckgegeben.

Anwendung

DISKSPACE() ist eine Environment-Funktion, die den verf.gbaren Speicherplatz eines Laufwerks in Byte ermittelt. Diese Funktion ist sehr hilfreich bei Kopier- oder Sortiervorg.ngen, um festzustellen, ob auch gen.gend freier Speicherplatz auf dem jeweiligen Laufwerk vorhanden ist. .ber DISKSPACE() in Verbindung mit RECSIZE() und RECCOUNT() lassen sich auch recht einfach automatische Backup-Prozeduren entwickeln.

DISKSPACE() ignoriert das von SET DEFAULT gesetzte Laufwerk.

Beispiele

. Dieses Beispiel demonstriert den Gebrauch von DISKSPACE(), um eine Datenbank auf ein anderes Laufwerk zu kopieren:

```
FUNCTION BackUp(cDatei, cLaufwerk)
LOCAL nGroesse
  //
  nGroesse := INT(RECSIZE() * LASTREC() + HAEDER() + 1)
  IF DISKSPACE(ASC(UPPER(cLaufwerk))) < nGroesse
    RETURN .F.
  ENDIF
  COPY TO (cLaufwerk + ":" + cDatei)
  //
RETURN .T.
```

Dateien

Bibliothek ist EXTEND.LIB, Quelldatei ist EXAMPLEC.C

See Also:

DISPOUT() Einen Wert auf dem Bildschirm anzeigen

Syntax

DISPOUT(<exp>) -> NIL

Parameter

<exp> ist der anzuzeigende Wert.

R.ckgabe

DISPOUT() besitzt keinen R.ckgabewert.

Anwendung

DISPOUT() ist eine einfache Ausgabefunktion, die einen Wert auf dem Bildschirm anzeigt. Sie folgt nicht der Redirektion mittels SET DEVICE. Die Ausgabe erfolgt immer auf den Bildschirm.

Beispiele

Das folgende Beispiel wurde aus dem Programm GETSYS.PRG entnommen und dient der Statusanzeige im "SCOREBOARD" des Bildschirms. Beachten Sie, wie die Bildschirmposition mit ROW() und COL() gesichert und anschlie.end mit SETPOS wiederhergestellt wird:

```
STATIC PROCEDURE ShowScoreboard
  LOCAL nRow, nCol

  IF (SET(_SET_SCOREBOARD ))
    nRow := ROW()
    nCol := COL()
    SETPOS(SCORE_ROW, SCORE_COL)
    DISPOUT( IF( SET( SET_INSERT), "Einfg.", "      ") )
    SETPOS( nRow, nCol )
  ENDIF
  RETURN
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

DOSERROR() Ermittelt die letzte DOS-Fehlernummer

Syntax

DOSERROR() -> nFehlerCode

R.ckgabe

Ein ganzzahliger, numerischer Wert, der den DOS-Fehlercode darstellt.

Anwendung

DOSERROR() ist eine Fehler-Funktion. Sie wird benutzt, um festzustellen, ob ein DOS-Fehler nach einer Operation, welche einen DOS-Fehler erzeugen kann, aufgetreten ist. Der Wert der Funktion bleibt so lange erhalten, bis eine andere DOS-Operation durchgef.hrt wird. Bei Low-level Dateifunktionen erzeugt FERROR() das gleiche Ergebnis wie DOSERROR.

Eine komplette Liste der m.glichen DOS-Fehlernummern ist im Anhang E des Handbuches Programmierung und Hilfsprogramme aufgef.hrt.

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

DOW() Ermittelt den Wochentag zu einem Datum

Syntax

DOW(<dDatum>) -> nWochentag

Parameter

<dDatum> ist ein Datumswert vom Typ Datum, von dem der Wochentag ermittelt werden soll.

R.ckgabewert

DOW() gibt den Wochentag als numerischen Wert zwischen 1 und 7 zur.ck. Der erste Tag der Woche ist 1 (Sonntag) , der letzte Tag der Woche ist 7 (Samstag). Wird DOW() ein leeres Datum (" / / ") .bergeben, ist der R.ckgabewert Null.

Anwendung

DOW() ist eine Datumskonvertierungs-Funktion, welche ein Datum in einen numerischen Wert umwandelt. Dieser Wert identifiziert den Tag innerhalb einer Woche. Sie ist verwendbar, um Berechnungen mit Datumswerten auf Wochentagsbasis durchzuf.hren. DOW() ist .hnlich der Funktion CDOW(), welche den Wochentagsnamen als Zeichenkette anstatt eines numerischen Wertes liefert.

Beispiele

. Die folgenden Zeilen zeigen den Einsatz von DOW() und den Unterschied zu CDOW():

```
? DATE()              // Resultat: 13.07.90
? DOW(DATE())         // Resultat: 6
? CDOW(DATE())        // Resultat: Freitag
? DOW(DATE() - 2)     // Resultat: 4
? CDOW(DATE() - 2)    // Resultat: Mittwoch
```

. Das folgende Beispiel zeigt eine Anwenderfunktion, die das Datum des vorangegangenen Montags als Ergebnis liefert:

```
FUNCTION LMontag(dDatum)
RETURN (dDatum - DOW(dDatum) + 2)
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

DTOC() Wandelt ein Datum in eine Zeichenfolge um

Syntax

DTOC(<dDatum>) -> cDatum

Parameter

<dDatum> ist das zu konvertierende Datum.

R.ckgabe

DTOC() erzeugt eine Zeichenfolge, die das Datum, entsprechend der aktuellen Datumsformat-Einstellung enth.lt. Das Standardformat ist mm/tt/jj. Ein Null-Datum liefert eine Zeichenkette mit Leerzeichen, welche in der L.nge durch die aktuelle Datumsformat-Einstellung bestimmt wird.

Anwendung

DTOC() ist eine Datumskonvertierungs-Funktion, die eingesetzt werden kann, wenn Sie formatierte Datumsdarstellungen .ber das SET DATE-Format als Zeichenkette ben.tigen (z.B. in LABEL FORM). Wenn ein spezielles Datumsformat ben.tigt wird, k.nnen Sie die Funktion TRANSFORM() benutzen.

Wenn in Indexausdr.cken ein Datum in Kombination mit einer Zeichenkette ben.tigt wird, verwenden Sie besser die Funktion DTOS() anstelle von DTOC(), um das Datum in eine Zeichenkette zu konvertieren.

Beispiele

Das Beispiel zeigt den generellen Gebrauch von DTOC():

```
? DATE()                          // Resultat: 13.07.90
? DTOC(DATE())                    // Resultat: "13.07.90"
? "Heute ist der " + DTOC(DATE())   // Resultat: "Heute ist der 13.07.90"
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

DTOS() Wandelt ein Datum in eine Zeichenfolge vom Format jjjjmmtt um

---

Syntax

DTOS(<dDatum>) -> cDatum

Parameter

<dDatum> ist der umzuwandelnde Datumswert.

R.ckgabe

DTOS() liefert eine acht Zeichen lange Zeichenkette im Format jjjjmmtt zur.ck. Ist <dDatum> ein leeres Datum, wird Zeichenfolge mit 8 Leerzeichen zur.ckgegeben. Das Format der R.ckgabe wird nicht durch die aktuelle Datumsformat-Einstellung (z.B. SET DATE) beeinflu.t.

Anwendung

DTOS() ist eine Datumskonvertierungs-Funktion und wird gebraucht, wenn in Indexausdr.cken ein Datum in Verbindung mit Zeichenketten verwendet werden soll. DTOS() konvertiert ein Datum zu einer Zeichenkette, welche mit beliebigen anderen Zeichenketten verkn.pft werden kann. Ein solches Ergebnis beh.lt die richtige Datumsfolge (Jahr, Monat und Tag) bei.

Beispiele

. Das folgende Beispiel zeigt DTOS() in Verbindung mit anderen Funtkionen:

```
? DATE()                  // Resultat: 01.09.90
? DTOS(DATE())            // Resultat: "19900901"
? LEN(DTOS(CTOD("")))     // Resultat: 8
```

. Das Beispiel zeigt, wie DTOS() verwendet werden kann, um einen gemischten Indexausdruck aus einer Zeichenfolge und einem Datum zu bilden:

```
USE Verkauf NEW
INDEX ON DTOS(VerkaufDat) + Artikel TO ArtBeweg
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

EMPTY() Ermittelt, ob das Ergebnis eines Ausdrucks leer ist

Syntax

EMPTY(<exp>) -> lLeer

Parameter

<exp> ist ein Wert beliebigen Datentyps.

R.ckgabe

EMPTY() gibt true (.T.) zur.ck, wenn <exp> einen leeren Wert enth.lt, andernfalls wird false (.F.) zur.ckgegeben. Ob ein Ausdruck ein leerer Wert ist, h.ngt auch von seinem Datentyp ab. Die folgende Tabelle gibt Aufschlu. dar.ber, wann ein Wert als leer gilt.

Tabelle 5-14: EMPTY() Liste leerer Werte

Datentyp	Inhalt
Array	Keine Elemente
Zeichen	Leerzeichen, Tabs, CR/LF oder ""
Numerisch	0
Datum	Null (CTOD(""))
Logisch	False (.F.)
Memo	Siehe Zeichen
NIL	NIL

Anwendung

EMPTY() ist eine vielseitige Abfragefunktion und findet daher eine Reihe von Verwendungen. Sie kann dazu benutzt werden, um festzustellen, ob ein Benutzer in einem GET-Objekt etwas eingegeben hat, bevor es in einer Datei abgelegt wird. Sie kann ebenso zur Feststellung benutzt werden, ob ein formaler Parameter nicht oder ohne Inhalt angegeben wurde. Desweiteren kann sie testen, ob ein Array Elemente enth.lt.

Beispiele

. Das Beispiel zeigt die Anwendung von EMPTY() auf verschiedene Datentypen:

```
? EMPTY(SPACE(5))    // Resultat: .T.
? EMPTY("")          // Resultat: .T.
? EMPTY(0)           // Resultat: .T.
? EMPTY(CTOD(""))    // Resultat: .T.
? EMPTY(.F.)         // Resultat: .T.
? EMPTY(NIL)         // Resultat: .T.
```

. Dieses Beispiel benutzt EMPTY(), um festzustellen, ob ein Benutzer eine Eingabe im ersten GET-Objekt gemacht hat, bevor die Eingabe in der Datenbank abgelegt wird:

```
USE Verkauf NEW
LOCAL cKunde := SPACE(15), nUmsatz := 0
@ 10, 10 GET cKunde
@ 11, 10 GET nUmsatz
READ
//
IF .NOT. EMPTY(cKunde)
  APPEND BLANK
  Verkauf->cKunde := cKunde
  Verkauf->Umsatz := nUmsatz
ENDIF
```

. Das Beispiel benutzt EMPTY() als Teil einer VALID-Bedingung, um die G.ltigkeit einer Benutzereingabe in einem GET-Objekt zu pr.fen:

```
LOCAL cKennung := SPACE(5)
@ 2, 5 SAY "Ihre Kennung: " GET cKennung VALID .NOT.EMPTY(cKennung)
READ
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

EOF() Ermittelt, ob das Dateiende erreicht ist

Syntax

EOF() -> lDateiEnde

R.ckgabe

EOF() liefert true (.T.), wenn versucht wird, den Datensatzzeiger hinter den letzten logischen Datensatz einer Datei zu setzen, andernfalls wird false (.F.) zur.ckgegeben. Ist keine Datenbank im aktuellen Arbeitsbereich ge.ffnet, liefert EOF() false (.F.). Enth.lt die aktuelle Datenbank keine Datens.tze, wird true (.T.) zur.ckgegeben.

Anwendung

EOF() ist eine Datenbankfunktion, die benutzt wird, um das Dateiende einer aktuellen Datenbank festzustellen, wenn der Datensatzzeiger vorw.rts bewegt wird. Einige Datenbank-Befehle k.nnen den Datensatzzeiger bewegen und EOF() setzen.

.blicherweise wird EOF() als Teil einer logischen Bedingung in einer DO WHILE-Schleife eingesetzt, in der eine sequentielle Bearbeitung von Datens.tzen vorgenommen wird. Dabei lautet die Bedingung sinnvollerweise NOT EOF(), damit die Schleife erst verlassen wird, wenn EOF() true (.T.) liefert.

EOF() und FOUND() werden oft gebraucht, wenn nach einem SEEK-, FIND- oder LOCATE-Befehl das Ergebnis getestet werden soll. In diesem Fall ist die Funktion FOUND() jedoch vorzuziehen.

Wenn EOF() true (.T.) liefert, steht der Datensatzzeiger auf LASTREC() + 1, unabh.ngig davon, ob SET FILTER aktiv oder SET DELETED auf ON gesetzt ist. Weitere Versuche, den Satzzeiger vorw.rts zu bewegen, bleiben bei dem gleichen Ergebnis, ohne einen Fehler hervorzurufen. Liefert EOF() einmal true (.T.), bleibt der Wert so lange erhalten, bis ein weiterer Versuch unternommen wird, den Satzzeiger zu bewegen.

Als Voreinstellung arbeitet EOF() im aktuellen Arbeitsbereich. Wenn EOF() auf andere Arbeitsbereiche angewendet werden soll, mu. ein Aliasbezeichner vorangestellt werden.

Beispiele

. Das folgende Beispiel zeigt den Einsatz von EOF() mit absichtlicher Satzzeigerbewegung .ber das Ende der Datei hinaus:

```
USE Verkauf
GO BOTTOM   // Zum letzten Datensatz
? EOF()     // Resultat: .F.
SKIP
? EOF()     // Resultat: .T.
```

. Das Beispiel benutzt einen Aliasbezeichner, um EOF() in anderen Arbeitsbereichen abzufragen:

```
USE Verkauf NEW
USE Kunden NEW
USE Rechnung NEW
? Verkauf->(EOF())
? Kunden->(EOF())
```

. Im letzten Beispiel wird EOF() benutzt, um eine sequentielle Datenbankbearbeitung abzubrechen, sobald das Dateiende erreicht ist:

```
USE Verkauf INDEX KundNr NEW
DO WHILE .NOT.EOF()
                                AlteKundenNr := Verkauf->KundenNr
                                UmsatzSumme := 0
                                DO WHILE (AlteKundenNr = Verkauf->KundenNr) .AND. .NOT.EOF()
.? Verkauf->KundenNr, Verkauf->Bemerkung, Verkauf->Umsatz
```

```
    .UmsatzSumme += Verkauf->Umsatz
    .SKIP
                                    ENDDO
                                    ? "Gesamtumsatz: ", UmsatzSumme
                                    ?
    ENDDO
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

ERRORBLOCK() Erzeugt einen Code-Block, welcher bei einem Laufzeitfehler ausgef.hrt wird

---

Syntax

ERRORBLOCK([<bFehlerBehandlung>]) -> bAktuelleFehlerBehandlung

Parameter

<bFehlerBehandlung> ist ein Code-Block, der immer dann ausgef.hrt wird, wenn zur Laufzeit ein Fehler auftritt. Wenn der Code-Block <bFehlerBehandlung> durchgef.hrt wird, bekommt er vom System ein Fehler-Objekt als Argument .bergeben.

R.ckgabe

ERRORBLOCK() liefert den Code-Block mit der aktuellen Fehlerbehandlung. Ist kein Fehlerbehandlungs-Block seit Programmstart angegeben, so liefert ERRORBLOCK() den Standard Fehlerbehandlungsblock.

Anwendung

ERRORBLOCK() ist eine Fehlerfunktion, die eine Fehlerbehandlungs-Routine definiert, welche bei Auftreten eines Laufzeitfehhlers ausgef.hrt wird. Die Fehlerbehandlungs-Routine ist als Code-Block in folgender Form ausgelegt:

```
{ |<objError>| <Ausdrucksliste>... }
```

Dabei ist <objError> ein Fehlerobjekt mit n.heren Informationen .ber den Fehler. Innerhalb der Code-Bl.cke k.nnen Nachrichten (messages), welche die Fehlerinformationen enthalten, an das Fehlerobjekt gesendet werden.Gibt der Code-Block true (.T.) zur.ck, wird die fehlerhafte Operation wiederholt; false (.F.) bedeutet, da. der Prozess fortgesetzt wird.

Der Fehlerbehandlungs-Codeblock kann eine Reihe von Ausdr.cken oder auch den Aufruf einer Anwenderfunktion (UDF) enthalten. Die Angabe einer Anwenderfunktion ist sinnvoller, da sie anstelle von Ausdr.cken auch Clipper-Kontrollbefehle enthalten kann; besonders dann, wenn nach einer BEGIN SEQUENCE-Anweisung .ber BREAK eine bestimmte Ausgangssituation erreicht werden soll.

Dies impliziert, da. der Fehlerbehandlungs-Block in Kombination mit den Clipper-Kontrollanweisungen BEGIN SEQUENCE ... BREAK ... END benutzt werden kann. Innerhalb des Fehlerbehandlungs-Blocks werden Ger.te-, Low-Level- und allgemeine Fehler behandelt, welche grunds.tzlich Wiederherstellungsmechanismen besitzen.

Ist keine <bFehlerBehandlung> durch ERRORBLOCK() angegeben und tritt w.hrend der Programmausf.hrung ein Fehler auf, wird ein Standard-Fehlerbehandlungsblock ausgef.hrt. Diese Fehlerbehandlung zeigt eine Fehlernachricht am Bildschirm, setzt den Wert in ERRORLEVEL auf 1 und verl..t das Programm wie .ber QUIT.

Nachdem ERRORBLOCK() den aktuellen Fehlerbehandlungsblock verl..t, ist es m.glich, diesen f.r weitere Operationen zu sichern, einen neuen Block zu definieren und sp.ter den gesicherten Block zu deaktivieren. Als wichtige Konsequenz, weshalb Fehlerbehandlungs-Routinen in Form von Code-Bl.cken spezifiziert sind, k.nnen diese leicht als Werte an Prozeduren und User Definded Functions .bergeben werden.

Weitere Informationen .ber die Struktur und Anwendung von Fehlerobjekten erhalten Sie im Abschnitt Fehler-Klassen des Kapitels Objektklassen.

Beispiele

Das folgende Programmbeispiel zeigt, wie ein Fehlerbehandlungsblock vordefiniert und sp.ter aufgerufen wird, wenn ein Fehler innerhalb von einem BEGIN SEQUENCE-Konstrukt auftritt:

```
LOCAL bFehlerBehandlung, bAlteBehandlung, objFehler
bFehlerBehandlung   := {|objFehler| FehlerBehandlung(objFehler) }
```

```
//
// aktuelle Fehlerbehandlung retten
bAlteBehandlung      := ERRORBLOCK(bFehlerBehandlung)
//
BEGIN SEQUENCE
  .
  . <Anweisungen>
  .
RECOVER USING objFehlerInfo // Fehler-Objekt von BREAK empfangen
  .
  . <Wiederherstellungs-Anweisungen>
  .
  END
ERRORBLOCK(bAlteBehandlung)
RETURN


FUNCTION FehlerBehandlung( objError )
  //
  // BREAK objError
RETURN NIL
```

Dateien

Bibliothek ist CLIPPER.LIB

ERRORLEVEL() Setzt den Clipper Beendigungs-Code

Syntax

ERRORLEVEL([<nReturnCode>]) -> nAktReturnCode

Parameter

<nReturnCode> ist der neue Beendigungs-Code, ein numerischer Wert zwischen Null und 255. Wird kein Wert angegeben, liefert ERRORLEVEL() den aktuellen Beendigungs-Code, ohne ihn zu beeinflussen.

R.ckgabe

ERRORLEVEL() liefert den aktuellen Beendigungs-Code, sofern dieser zuvor gesetzt worden ist. Wurde der Beendigungs-Code nicht gesetzt, wird Null zur.ckgegeben.

Anwendung

ERRORLEVEL() erf.llt zwei Aufgaben. Sie k.nnen den aktuellen Beendigungs-Code ermitteln und neu setzen. Normalerweise wird der Beendigungs-Code von einem untergeordneten Prozess gesetzt, damit ein .bergeordneter Prozess den Beendigungsstatus .berpr.fen kann.

Typisch ist die Anwendung zwischen verschiedenen Applikationen und DOS. Ein in einer Clipper-Applikation gesetzter Beendigungs-Code kann in einer DOS-Stapelverarbeitung .ber ERRORLEVEL angefragt oder .ber die DOS-Funktion 4Dh von einer weiteren Anwendung abgeholt werden.

In Clipper wird der Beendigungs-Code immer dann auf 1 gesetzt, wenn das Programm mit einem schweren Fehler beendet worden ist. Im anderen Fall ist der Beendigungs-Code 0, oder er hat den Wert, der zuletzt mit ERRORLEVEL() gesetzt worden ist.

Hinweise

. ERRORLEVEL() wird nicht ver.ndert, wenn Sie mit der RUN-Anweisung einen DOS-Befehl ausf.hren. Um von solch untergeordneten Prozessen den Beendigungs-Code zu erfahren, m.ssen Sie ein C- oder Assemblerprogramm einbinden, das .ber die Funktion 4Dh des Interrupts 21 den jeweiligen Beendigungs-Code abfragt. Lesen Sie dazu bitte auch im DOS-Handbuch nach.

Beispiele

. Das folgende Beispiel sichert den aktuellen Beendigungs-Code und setzt einen neuen Wert:

```
CodeAlt := ERRORLEVEL()        // Holen des alten Return-Codes
ERRORLEVEL(1)                  // Setzen des Return-Codes auf 1
```

. Das n.chste Beispiel benutzt ERRORLEVEL(), um in einem .bergeordneten Prozess auf einen m.glichen Fehler zu reagieren:

```
#define DateiFehlt 255
#define DateiFehler 254

IF .NOT. FILE("system.dbf")
                                ? "Schwerer Fehler: Systemdatei kann nicht gefunden werden !"
                                ? "Programm wurde abgebrochen"
                                ERRORLEVEL(DateiFehlt)
                                QUIT
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

EVAL() Wertet einen Code-Block aus

Syntax

EVAL(<bBlock>, [<ArgumentListe>]) -> xErgebnisWert

Parameter

<bBlock> ist der zu evaluierende Code-Block.

<ArgumentListe> ist die Liste der Argumente, die an den Code-Block .bergeben werden.

R.ckgabe

EVAL() gibt das Ergebnis der letzten Berechnung eines Code-Blocks zur.ck. Er kann einen Wert beliebigen Typs zur.ckgeben.

Anwendung

EVAL() ist eine Code-Block-Funktion. Sie ist im Clipper-System die Basisfunktion, die Code-Bl.cke ausf.hrt. Ein Code-Block ist ein spezieller Datenwert, welcher auf ein St.ck compiliertem Code zeigt. N.here Informationen zu Code-Bl.cken erfahren Sie im Kapitel Grundkonzepte des Referenzhandbuches.

Um Code-Bl.cke auszuf.hren oder zu evaluieren, wird EVAL() mit einem Code-Block und beliebigen Argumenten aufgerufen. Die Argumente werden an den Code-Block .bergeben, wenn dieser ausgef.hrt wird. Code-Bl.cke k.nnen eine Reihe von Ausdr.cken enthalten, wenn diese jeweils durch ein Komma getrennt werden. Nach der Ausf.hrung eines Code-Blocks, wird der Wert zur.ckgegeben, der als letzter Ausdruck angegeben wurde.

Ein Code-Block wird w.hrend des Clipper-Compilierlaufes .bersetzt. Er kann auch w.hrend der Laufzeit compiliert werden, wenn er aus einer Zeichenfolge ausgef.hrt wird. Dies geschieht durch den Makro-Operator (&).

EVAL wird oft dazu benutzt, um Iterationsfunktionen zu bilden. Dies sind Funktionen, die einen Code-Block f.r jeden Teil einer Datenstruktur durchlaufen. AEVAL(), ASORT(), ASCAN(), und DBEVAL() sind solche iterativen Funktionen. AEVAL() f.hrt beispielweise einen Code-Block f.r jedes Element eines Arrays aus.

Beispiele

. Das Beispiel erzeugt einen Code-Block, welcher einen Wert um 1 erh.ht:

```
bBlock := {| nArg |  nArg + 1 }
? EVAL(bBlock, 1)                    // Resultat: 2
```

. Das n.chste Beispiel demonstriert, wie ein Code-Block durch den Gebrauch des Makro-Operators (&) zur Laufzeit compiliert wird:

```
bBlock := &( "{|nArg| nArg + 1 }" ) // Compiliert einen String
? EVAL(bBlock, 1)                   // Berechnung des Code-Blocks
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

EXP() Berechnung von e^x

Syntax

EXP(<nExponent>) -> nNatLog

Parameter

<nExponent> ist der nat.rliche Logarithmus, der f.r den Wert berechnet werden soll.

R.ckgabe

EXP() gibt einen numerischen Wert zur.ck, der das Ergebnis von e hoch <nExponent> ist.

Anwendung

EXP() ist eine mathematische Funktion und berechnet den Wert von ex, wobei e der nat.rliche Logarithmus und x der Exponent ist. Der maximale Wert f.r <nExponent> ist 45, bevor ein numerischer .berlauf eintritt. EXP() ist die inverse Funktion von LOG().

Die Anzahl der Nachkommastellen h.ngt von der Einstellung der Anweisung SET DECIMALS und nicht von SET FIXED ab.

Beispiele

Dieses Beispiel demonstriert einige Anwendungen von EXP():

```
? EXP(1)              // Resultat: 2.72
SET DECIMALS TO 10
? EXP(1)              // Resultat: 2.7182818285
? LOG(EXP(1))         // Resultat: 1,0000000000
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

FCLOSE() Schlie.t eine ge.ffnete Bin.r-Datei und sichert den Dateipuffer

---

Syntax

FCLOSE(<nDateiKennung>) -> lGeschlossen

Parameter

<nDateiKennung> ist die Datei-Kennung (Handle), anhand der die Datei zuvor mit FOPEN() oder FCREATE() ge.ffnet wurde.

R.ckgabe

Es wird .T. zur.ckgegeben, wenn die Datei ordnungsgem.. geschlossen und die Dateipuffer auf die Disk geschrieben wurden. Bei einem Fehler wird .F. zur.ckgegeben.

Anwendung

FCLOSE(), eine Low-Level Dateifunktion, schlie.t eine Datei und schreibt die Dateipuffer, die dieser zugeordnet sind, auf Festplatte oder Diskette. Tritt dabei ein Fehler auf, wird der Wert false (.F.) zur.ckgegeben. Um die genaue Ursache des Fehlers zu erfahren, kann die Funktion FERROR() herangezogen werden. Zum Beispiel erzeugt FCLOSE() false, wenn eine nicht bekannte Datei-Kennung angegeben wird. In diesem Falle liefert FERROR() den Wert 6 "invalid handle". Eine komplette Liste der Fehlernummern finden Sie unter FERROR().

Warnung

Diese Funktion erlaubt den Dateizugriff auf Systemebene (Low-Level). Sie sollte nur mit .u.erster Vorsicht und genauen Kenntnissen des Betriebssystems angewendet werden.

Beispiele

Das folgende Beispiel schlie.t eine mit FCREATE() erzeugte Datei und schreibt eine Fehlermeldung, wenn die Operation fehlschl.gt:

```
#include "FileIO.ch"
//
nKennung := FCREATE("test.dat", FC_NORMAL)
IF .NOT.FCLOSE(nKennung)
                                ? "Fehler beim Schlie.en der Datei. Fehlernummer: ", FERROR()
ENDIF
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

FCOUNT() Ermittelt die Anzahl der Felder in der aktuellen Datenbank

---

Syntax

FCOUNT() -> nAnzahl

R.ckgabe

FCOUNT() gibt die Anzahl der Felder der aktuellen Datenbank als numerischen, ganzzahligen Wert zur.ck. Ist keine Datenbank ge.ffnet, wird der Wert Null zur.ckgegeben.

Anwendung

FCOUNT() ist eine Datenbankfunktion. Sie ist immer dann zu gebrauchen, wenn Sie Anwendungen schreiben, in denen Sie keine Kontrolle .ber die Struktur einer Datenbank haben, zum Beispiel bei der Bearbeitung beliebiger Datenbanken. Dazu geh.ren haupts.chlich Import-, Export- und Reportfunktionen. In der Regel wird FCOUNT() dazu verwendet, eine FOR...NEXT- oder DO WHILE-Struktur aufzubauen, innerhalb der ein einzelnes Feld bearbeitet werden soll.

Als Voreinstellung arbeitet FCOUNT() auf dem aktuellen Arbeitsbereich. Die Funktion kann aber auch durch Voranstellen eines Aliasbezeichners auf andere Arbeitsbereiche angewendet werden.

Beispiele

. Das folgende Beispiel zeigt FCOUNT(), um die Anzahl der Felder des aktuellen und eines unselektierten Arbeitsbereiches zu ermitteln:

```
USE Verkauf NEW
USE Kunden NEW
? FCOUNT()                // Resultat: 5
? Verkauf->(FCOUNT())     // Resultat: 8
```

. Das n.chste Beispiel benutzt FCOUNT(), um ein Array zu deklarieren, welches mit Feldinformationen gef.llt wird:

```
LOCAL aFelder := ARRAY(FCOUNT())
AFIELDS(aFelder)
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

FCREATE() Legt eine Bin.r-Datei an und/oder k.rzt diese auf die L.nge Null

---

Syntax

FCREATE(<cDateiName>, [<nAttribut>]) -> nDateiKennung

Parameter

<cDateiName> ist der Name der anzulegenden Datei. Existiert bereits eine Datei mit gleichem Namen, wird diese auf Null-L.nge gek.rzt.

<nAttribut> ist eines der Dateiattribute, die in der folgenden Tabelle gezeigt werden. Der Standardwert betr.gt 0.

Tabelle 5-15: FCREATE() Dateiattribute

Wert	FileIO.ch	Attribut	Beschreibung
0	FC_NORMAL	Normal	Lesen/Schreiben (Voreinstellung)
1	FC_READONLY	Read-only	Nur Lesen
2	FC_HIDDEN	Hidden	Versteckte Datei
4	FC_SYSTEM	System	Systemdatei

R.ckgabe

FCREATE() gibt die Nummer des DOS-Handles der entsprechenden Datei als numerischen Wert zwischen Null und 65.535 zur.ck. Tritt ein Fehler auf, wird -1 zur.ckgegeben und FERROR() liefert eine DOS-Fehlernummer.

Anwendung

FCREATE ist eine Low-Level Dateifunktion zum Anlegen einer neuen oder zum .ffnen und Leeren einer bestehenden Datei. Wenn <cDateiNname> nicht existiert, wird sie neu angelegt und gleichzeitig zum Schreiben ge.ffnet. Existiert die Datei bereits, so wird sie ge.ffnet und der Inhalt geleert. Wenn sie nicht zum Schreiben ge.ffnet werden kann, liefert FCREATE() -1 als R.ckgabe und FERROR() wird auf den entsprechenden Fehlerwert gesetzt.

Wurde mit FCREATE() eine neue Datei erzeugt, so ist sie im DOS-OPEN-MODE 2 er.ffnet worden (kompatibler SHARE Modus, Lesen und Schreiben erlaubt). Das in <nAttribut> definierte Dateiattribut wird beim Schlie.en der Datei ben.tigt, um eine Datei anlegen zu k.nnen, die dann nur zum Lesen ge.ffnet werden kann. Eine Liste der Zugriffsm.glichkeiten k.nnen Sie der Beschreibung von FOPEN() entnehmen.

Um weitere Dateioperationen mit dieser Datei durchzuf.hren, mu. die Dateikennung (Handle) in einer Variablen gespeichert werden.

Wie auch die anderen Low-Level Dateifunktionen ignoriert FCREATE() die Anweisungen SET PATH und SET DEFAULT. Die Datei wird immer im aktuellen Verzeichnis angelegt, es sei denn, der Pfad wird explizit mit angegeben.

Warnung

Diese Funktion erlaubt den Dateizugriff auf der Systemebene (Low-Level). Sie sollte nur mit .u.erster Vorsicht und genauen Kenntnissen des Betriebssystems angewendet werden.

Beispiele

Dies ist ein einfaches Beispiel, in dem eine Datei mit dem Namen "Test.dat" erzeugt und im Modus Lesen/Schreiben ge.ffnet wird:

```
#include "FileIO.ch"
//
IF (nKennung := FCREATE("Test.dat", FC_NORMAL)) = -1
  ? "Datei kann nicht angelegt werden:", FERROR()
  BREAK
ELSE
  FWRITE(nKennung, "Hallo, hier steht etwas")
  FCLOSE(nKennung)
ENDIF
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

FERASE() L.scht eine Datei vom Laufwerk

Syntax

FERASE(<cDateiName>) -> nAusgef.hrt

Parameter

<cDateiName> ist der Name der Datei, die gel.scht werden soll. Der Dateiname mu. einschlie.lich Dateinamenserweiterung angegeben werden und kann optional Laufwerks- und Verzeichnisangaben enthalten.

R.ckgabe

FERASE() liefert -1, wenn die Funktion nicht durchgef.hrt werden konnte, andernfalls 0. Bei 1 kann mit FERROR() die Fehlerursache festgestellt werden.

Anwendung

FERASE() ist eine Low-Level Dateifunktion, mit der eine Datei vom Laufwerk gel.scht werden kann. FERASE ist identisch mit dem Befehl ERASE, liefert allerdings einen R.ckgabewert, der in einem Ausdruck verwendet werden kann. Beim Aufruf von FERASE() wird nur die Datei <cDateiName> im aktuellen oder, wenn angegeben, im spezifizierten Verzeichnis gel.scht. Wie auch bei anderen Low-Level Dateifunktionen werden die Einstellungen von SET DEFAULT und SET PATH ignoriert.

Warnung

Dateien, die mit FERASE() gel.scht werden sollen, m.ssen geschlossen sein.

Beispiele

. Das Beispiel zeigt, wie mit FERASE() eine Reihe von Dateien, die .ber Platzhalter angegeben sind, gel.scht werden:

```
#include "Directry.ch"
AEVAL(DIRECTORY("*.BAK"), {|aDatei|  FERASE(aDatei[F_NAME]) })
```

. Das folgende Beispiel l.scht eine Datei im Hauptverzeichnis von Laufwerk A und zeigt eine Fehlernachricht, wenn die Operation fehlgeschlagen ist:

```
IF FERASE("A:\Test.dat") = -1
  ? "Datei nicht gel.scht ! Fehler :", FERROR()
  BREAK
ENDIF
```

Dateien Bibliothek ist CLIPPER.LIB

FERROR() Ermittelt den Fehler-Code einer Low-Level Dateifunktion

---

Syntax

FERROR() -> nFehlerCode

R.ckgabe

FERROR() gibt den DOS-Fehlercode der letzten Dateioperation als numerischen Wert zur.ck. Ist kein Fehler aufgetreten, wird Null zur.ckgegeben.

Tabelle 5-16: FERROR() R.ckgabewerte

Fehler	Bedeutung
0	Kein Fehler aufgetreten
2	Datei nicht gefunden
3	Verzeichnis nicht gefunden
4	Zu viele ge.ffnete Dateien
5	Zugriff verweigert
6	Ung.ltiger DOS-Handle
8	Kein Speicher mehr vorhanden
15	Ung.ltige Laufwerksangabe
19	Versuch, auf eine schreibgesch.tzte Diskette zu schreiben
21	Laufwerk nicht bereit
23	CRC Datenfehler (Checksumme fehlerhaft)
29	Schreibfehler
30	Lesefehler
32	Ung.ltiger Mehrfachzugriff
33	Mehrfachzugriff gesperrt

Anwendung

FERROR() ist eine Low-Level Dateifunktion. Sie dient dazu, eine eventuelle Fehlerursache eines vorangegangenen Low-Level Dateizugriffs zu ermitteln. Das schlie.t die Funktionen FCLOSE(), FCREATE(), FERASE(), FOPEN(), FREAD(), FREADSTR() und FRENAME() ein. FERROR() beh.lt seinen Wert, bis erneut eine Low-Level Funktion ausgef.hrt wird.

Warnung

Diese Funktion erlaubt Dateizugriffe auf der Systemebene (Low-Level). Sie sollte nur mit .u.erster Vorsicht und genauen Kenntnissen des Betriebssystems angewendet werden.

Beispiele

Das folgende Beispiel .berpr.ft das Anlegen einer neuen Datei und gibt eine Fehlermeldung aus, wenn die Operation fehlschl.gt:

```
#include "FileIO.ch"
//
nKennung := FCERATE("Temp.txt", FC_NORMAL)
IF FERROR() <> 0
? "Datei nicht angelegt! DOS-Fehler : ", FERROR()
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

FIELD() Ermittelt einen Feldnamen der aktuellen Datenbank

Syntax

FIELD(<nPosition>) -> cFeldname

Parameter

<Position> ist die Position des Feldes in der Datenbankstruktur.

R.ckgabe

FIELD() gibt den Namen des Datenbankfeldes in einer Zeichenkette in Gro.buchstaben zur.ck. Ist die angegebene Feldposition nicht vorhanden, wird eine leere Zeichenkette ("") zur.ckgegeben.

Anwendung

FIELD() ist eine Datenbankfunktion, welche den .ber <nPosition> indizierten Feldnamen einer aktuellen Datenbank als Zeichenkette liefert. Sie wird normalerweise in Applikationen angewendet, in denen die Feldnamen nicht bekannt sind. Wenn Informationen von mehr als einem Feld ben.tigt werden, kann AFIELDS() zur Erzeugung eines Datenstruktur-Arrays, oder COPY STRUCTURE EXTENDED zur Erzeugung einer Datenstruktur-Datei benutzt werden.

Als Voreinstellung verwendet FIELD() den aktuellen Arbeitsbereich. Die Funktion kann ebenfalls durch Voranstellen eines Aliasbezeichners auf andere Arbeitsbereiche angewendet werden.

Beispiele

. Das Beispiel zeigt FIELD() im Zusammenhang mit anderen Funktionen:

```
USE verkauf
? FIELD(1)          // Resultat: BRANCHE
? FCOUNT()          // Resultat: 5
? LEN(FIELD(0))     // Resultat: 0
? LEN(FIELD(40))    // Resultat: 0
```

. Das folgende Beispiel benutzt FIELD(), um den Namen und Typ eines jeden Feldes aus der Datei Kunden aufzulisten:

```
USE kunden NEW
FOR nFeld := 1 TO FCOUNT()
                                ? PADR(FIELD(nFeld), 10), VALTYPE( &(FIELD(nFeld) )
NEXT
```

. Das n.chste Beispiel zeigt, wie Aliasnamen verwendet werden, um Feldnamen nichtselektierter Arbeitsbereiche zu ermitteln:

```
USE Verkauf NEW
USE Kunden NEW
USE Rechnung NEW
//
? Verkauf->(FIELD(1))       // Resultat: verkauf_nr
? Kunden->(FIELD(1))        // Resultat: kund_nr
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

FIELDBLOCK() Liefert einen SET/GET-Code-Block f.r eine Feldvariable

---

Syntax

FIELDBLOCK(<cFeldName>) -> bFeldBlock

Parameter

<cFeldName> ist der Name des Feldes, f.r das ein Code-Block erzeugt werden soll.

R.ckgabe

FIELDBLOCK() liefert einen Code-Block f.r eine Feldvariable. Wird dieser Block ausgef.hrt, liefert oder .bergibt er einen Wert von einem oder an ein Datenbankfeld. Existiert im aktuellen Arbeitsbereich kein Feld <cFeldname>, liefert die Funktion NIL.

Anwendung

FIELDBLOCK() ist eine Datenbankfunktion, die einen Code-Block erzeugt. Wird der Code-Block mit einen Paramerter ausgef.hrt, Weist der Code-Block diesen Wert einer Feldvariablen zu. Wird die der Code-Block ohne Parameter ausgef.hrt, liefert er den Wert der Feldvariablen <cFeldName>.

Hinweise

. **Arbeitsbereich:** Der von der FIELDBLOCK() zur.ckgegebenen Codeblock setzt oder holt den Wert eines Feldes immer dann, wenn er ausgef.hrt wird. Er bezieht sich immer auf den Feldnamen des Arbeitsbereich, der zur Zeit der Ausf.hrung aktiv ist.

Beispiele

. Ein Beispiel mit zwei Arbeitsbereichen, in denen Dateien mit zwei gleichen Feldnamen existieren:

```
SELECT Kunden
Kunden->Name := "M.ller"
SELECT Verkauf
Verkauf->Name := "Schmitz"

bName := FIELDBLOCK("Name")

SELECT Kunden
? EVAL(bName)          // Ergebnis "M.ller"
SELECT Verkauf
? EVAL(bName)          // Ergebnis "Schmitz"
EVAL(bName,"Meier")    // Feld neu zuweisen
? EVAL(bName)          // Ergebnis "Meier"
```

See Also:

FIELDGET() Den Wert einer Feldvariablen ermitteln, ohne den Feldnamen zu kennen

---

Syntax

FIELDGET(<nFeld>) -> FeldWert

Parameter

<nFeld> ist die numerische Position der Feldvariable innerhalb der Datensatzstruktur.

R.ckgabe

FIELDGET() liefert den Inhalt des Datenbankfeldes. Wenn keine g.ltige Feldposition angegeben wurde oder keine Datenbank im aktuellen Arbeitsbereich ge.ffnet ist, liefert die Funktion NIL.

Anwendung

FIELDGET() ist eine Datenbankfunktion, die den Wert eines Datenbankfeldes .ber die Feldposition innerhalb der Datensatzstruktur ermittelt, ohne da. der Feldname bekannt sein mu.. Diese Funktion erlaubt unter anderem den Zugriff auf unbekannte Datenbankfelder ohne Verwendung von Makros.

Beispiele

Das folgende Beispiel vergleicht den Einsatz der Funktion mit einer funktionsgleichen Routine, welche einen Makro-Operator verwendet:

```
LOCAL nFeld := 1, Feldwert, Feldname
USE Kunden NEW

// Verwendung mit Makro-Operator
Feldname := FIELDNAME( nFeld )
Feldwert := &Feldname

// Verwendung von FIELDGET()
Feldwert := FIELDGET( nFeld )
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

FIELDPUT() Einer Feldvariable .ber Feldposition einen Wert zuweisen

---

Syntax

FIELDPUT(<nFeld>, <exp>) -> expWert

Parameter

<nFeld> ist die Feldposition innerhalb der aktuellen Datensatzstruktur.

<exp> ist der Wert, der dem Datenbankfeld zugewiesen werden soll. Der Datentyp mu. mit dem Feldtyp .bereinstimmen.

R.ckgabe

FIELDPUT() liefert den, der Feldvariablen zugewiesenen Wert zur.ck. Wurde keine g.ltige Feldposition angegeben oder ist im aktuellen Arbeitsbereich keine Datenbank ge.ffnet, wird NIL zur.ckgegeben.

Anwendung

FIELDPUT ist eine Datenbankfunktion, mit der einem Datenfeld ein Wert .ber die Feldposition innerhalb der Datensatzstruktur zugewiesen wird, ohne da. der Name des Feldes bekannt sein mu.. Diese Funktion gestattet die Wert-zuweisung auch auf unbekannte Datenbankfelder, ohne Verwendung von Makros.

Beispiele

Das folgende Beispiel vergleicht den Einsatz der Funktion mit einer funk-tionsgleichen Routine, welche einen Makro-Operator verwendet:

```
LOCAL nFeld := 1, Feldwert := "xx" , Feldname
USE Kunden NEW

// Verwendung mit Makro-Operator
Feldname := FIELDNAME( nFeld )
&Feldname:= Feldwert

// Verwendung von FIELDGET()
FIELDGET( nFeld, Feldwert )
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

FIELDBWLOCK() Liefert einen SET/GET-Code-Block f.r eine Feldvariable
eines spezifizierten Arbeitsbereiches

Syntax

FIELDWBLOCK(<cFeldName>, <nArbeitsbereich>) -> bFeldBlock

Parameter

<cFeldName> ist der Name des Feldes, f.r das ein Code-Block erzeugt werden soll.

<nArbeitsbereich> ist der numerische Arbeitsbereich, in der der Feldname angegeben ist.

R.ckgabe

FIELDWBLOCK() liefert einen Code-Block f.r eine Feldvariable. Wird dieser Block ausgef.hrt, liefert oder .bergibt er einen Wert von einem oder an ein Datenbankfeld. Existiert im aktuellen Arbeitsbereich kein Feld <cFeldname>, liefert die Funktion NIL.

Anwendung

FIELDWBLOCK() ist eine Datenbankfunktion, die einen Code-Block erzeugt. Wird der Code-Block mit einen Parameter ausgef.hrt, weist der Code-Block diesen Wert einer Feldvariablen zu. Wird der Code-Block ohne Parameter ausgef.hrt, liefert er den Wert der Feldvariablen <cFeldName>.

Hinweise

. Die Funktion FIELDWBLOCK() ist vergleichbar mit der FIELDBLOCK(), bezieht jedoch auf den angegeben Arbeitsbereich.

See Also:

FILE() Ermittelt die Existenz einer Datei im Clipper-Verzeichnis

---

Syntax

FILE(<cDateiName>) -> lExistent

Parameter

<cDateiName> ist die Dateispezifikation der Datei, nach der im voreingestellten Clipper-Pfad (SET PATH) und -Verzeichnis (SET DEFAULT) gesucht wird. Sie k.nnen bei der Angabe des Dateinamens sowohl die Jokerzeichen "*" und "?" benutzen, als auch das Laufwerk und Verzeichnis angeben. Die Dateinamenserweiterung mu. angegeben werden.

R.ckgabe

FILE() gibt true (.T.) zur.ck, wenn eine .ber <cDateiName> angegebene Datei existiert, false (.F.), wenn keine Datei gefunden wird.

Anwendung

FILE() ist eine Environment-Funktion und wird benutzt, um festzustellen, ob eine angegebene Datei existiert. Bei der Ausf.hrung von FILE() wird zun.chst im aktuellen Laufwerk und Verzeichnis gesucht. Ist die Datei nicht gefunden und wurde kein Verzeichnis angegeben, wird in den Verzeichnissen der Anweisung SET PATH und dann in den Verzeichnissen des DOS-Befehls PATH gesucht. Beachten Sie, da. die als hidden und system spezifizierten Dateien nicht ber.cksichtigt werden.

Beispiele

In diesem Beispiel wird nach der Datei "Verkauf.dbf" in einem Verzeichnis gesucht, das nicht das aktuelle Clipper-Verzeichnis ist:

```
? FILE("Verkauf.dbf")              // Resultat: .F.
? FILE("\apps\dbf\Verkauf.dbf")    // Resultat: .T.
SET PATH TO \APPS\DBF
? FILE("Verkauf.dbf")              // Resultat: .T.
SET PATH TO
SET DEFAULT TO \apps\dbf
? FILE("Verkauf.dbf")              // Resultat: .T.
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

FKLABEL()* Ermittelt den Namen einer Funktionstaste

Syntax

FKLABEL(<nFunktionstaste>) -> cTastenName

Parameter

<nFunktionstaste> ist ein Wert von 1-40, der die Funktionstasten [F1] bis [F10] alleine und in Verbindung mit [F9], [Strg] und [Alt] kennzeichnet.

R.ckgabe

FKLABEL() liefert den Namen einer Funktionstaste, die .ber <nFunktionstaste> angegeben wurde, als Zeichenkette. Ist der angegebene Parameter kleiner 1 oder g..er 40, wird immer eine leere Zeichenkette ("") zur.ckgegeben.

Anwendung

FKLABEL() ist eine Kompatibilit.tsfunktion und nur dazu da, die Funktion FKLABEL() aus sprach.hnlichen Interpretern zu unterst.tzen. Die Eigenschaft dieser Funktion soll die Portierung auf andere Systeme erleichtern, aber die Anwendung ist eingeschr.nkt.

Grunds.tzlich wird der Gebrauch dieser Funktion nicht empfohlen und in Clipper auch nicht ben.tigt. Die Funktionstasten sind gekennzeichnet mit Fn, wobei n im Bereich von 1 bis 40 liegt und direkt mit dem Parameter <nFunktionstaste> korrespondiert.

Dateien

Bibliothek ist EXTEND.LIB, Quelldatei ist Examplep.prg

See Also:

FKMAX()* Liefert die Anzahl der Funktionstasten als Konstante

---

Syntax

FKMAX() -> nFunktionstasten

R.ckgabe

FKMAX() liefert einen konstanten Wert von 40 zur.ck.

Anwendung

FKMAX() ist eine Kompatibilit.tsfunktion und nur dazu da, die Funktion FKMAX() aus sprach.hnlichen Interpretern zu unterst.tzen. Die Eigenschaft dieser Funktion soll die Portierung auf andere Systeme erleichtern, aber die Anwendung ist eingeschr.nkt.

Grunds.tzlich wird der Gebrauch dieser Funktion nicht empfohlen und in Clipper auch nicht ben.tigt. Sie liefert lediglich einen konstanten Wert von 40. Dieser Wert kann in Clipper leichter und effizienter .ber eine #define-Anweisung implementiert werden.

Dateien

Bibliothek ist EXTEND.LIB, Quelldatei ist Examplep.prg

See Also:

FLOCK() Sperrt eine Datenbank f.r die Mehrfachbenutzung

Syntax

FLOCK() -> lAusgef.hrt

R.ckgabe

FLOCK() gibt true (.T.) zur.ck, wenn die im aktuellen Arbeitsbereich .ber USE ge.ffnete Datenbank gesperrt werden konnte, ansonsten liefert FLOCK() false (.F.). Weitere Informationen sind im Kapitel Netzwerkprogrammierung im Handbuch Programmierung und Hilfsprogramme enthalten.

Anwendung

FLOCK() ist eine Datenbankfunktion, welche in einer Netzwerkumgebung verhindert, da. andere Benutzer in eine Datenbank schreiben, so lange die Sperrung nicht aufgehoben wird. Trotz der Sperrung ist der Lesezugriff immer erlaubt.

FLOCK() ist .hnlich den Anweisungen USE...EXCLUSIVE und RLOCK(). W.hrend USE...EXCLUSIV Datenbanken im Alleinzugriff .ffnet (kein anderer Benutzer ist in der Lage, die Datenbank zu .ffnen), sorgt RLOCK() daf.r, da. ein einzelner Datensatz f.r Schreibzugriffe gesperrt wird. FLOCK() ist daher in der Mitte dieser beiden Befehle anzusiedeln, da anderen Benutzern nicht verweigert wird, die Datenbank zu .ffnen, gleichzeitig aber jeder Schreibzugriff unterbunden wird.

Generell wird FLOCK() immer dann benutzt, wenn eine Operation ausgef.hrt wird, welche die gesamte Datenbank betrifft. Dazu geh.ren Anweisungen, die einen Bereich einer Datenbank betreffen oder mit einer Bedingung (DELETE oder REPLACE ALL) verbunden sind.

Tabelle 5-17: FLOCK() Anweisungen, die FLOCK() ben.tigen

Anweisung	Modus
APPEND FROM	FLOCK() oder USE...EXCLUSIVE
DELETE (mehrere S.tze)	FLOCK() oder USE...EXCLUSIVE
RECALL (mehrere S.tze)	FLOCK() oder USE...EXCLUSIVE
REPLACE (mehrere S.tze)	FLOCK() oder USE...EXCLUSIVE
UPDATE ON	FLCOK() oder USE...EXCLUSIVE

F.r jeden Aufruf von FLOCK() wird ein Versuch unternommen, die Datei zu sperren, und ein entsprechender logischer Wert zur.ckgegeben. Die Sperrung erfolgt nicht, wenn ein anderer Benutzer entweder die gesamte Datenbank oder einen einzelnen Satz gesperrt hat. Ist FLOCK() erfolgreich gewesen, besteht der Schreibschutz f.r andere Benutzer so lange, bis mit einer der Anweisungen UNLOCK, CLOSE oder RLOCK() die Sperrung wieder aufgehoben wird.

Als Voreinstellung arbeitet FLOCK() im aktuellen Arbeitsbereich. Die Funktion kann durch Voranstellen eines Aliasbezeichners auch auf nichtselektierte Arbeitsbereiche angewendet werden.

Hinweise

SET RELATION: Clipper f.hrt nicht automatisch auch ein FLOCK() in den Arbeitsbereichen durch, die mit dem zu sperrenden durch SET RELATION verkn.pft sind. UNLOCK hat ebenfalls keinerlei Effekt auf verkn.pfte Dateien.

Beispiele

. Dieses Beispiel zeigt die Verwendung von FLOCK() bei einem Update der Datenbank Inventar.dbf:

```
USE Inventar NEW
IF FLOCK()
REPLACE ALL Preis WITH Preis * 1.1
ELSE
? "Die Bearbeitung kann nicht gestartet werden."
ENDIF
```

. Das n.chste Beispiel zeigt die Verwendung des Aliasnamens, um eine Datei in einem nichtselektierten Arbeitsbereich zu sperren:

```
USE Verkauf NEW
USE Kunden NEW
IF Verkauf->(FLOCK())
? "Die Datenbank -Verkauf- wurde gesperrt"
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

FOPEN() .ffnet eine Bin.r-Datei

Syntax

FOPEN(<cDateiName>, [<nModus>]) -> nDateiKennung

Parameter

<cDateiName> ist der Name der Datei, dem optional auch Laufwerksbezeichner und Pfadangaben vorangestellt werden k.nnen.

<nModus> ist der gew.nschte DOS-Zugriffsmodus, mit dem die Datei ge.ffnet werden soll. Die verschiedenen Zugriffsarten werden in der Tabelle weiter unten gezeigt. Der Vorgabewert ist Null (nur Lesen).

Tabelle 5-18: FOPEN() Zugriffsmodus f.r Bin.rdateien

Modus	Operation
0	Nur Lesen
1	Nur Schreiben
2	Lesen und Schreiben

R.ckgabe

FOPEN() liefert eine von DOS vergebene Dateikennung, die im Bereich von 0 bis 65.535 liegen kann, als ganzzahligen, numerischen Wert. Tritt bei dem Versuch, die Datei zu .ffnen, ein Fehler auf, ist der R.ckgabewert -1.

Anwendung

FOPEN() ist eine Low-Level Dateifunktion, die eine bestehende Datei zum Lesen und Schreiben, angegeben durch <nModus>, .ffnet. Wenn beim .ffnen der Datei ein Fehler auftritt, kann FERROR() benutzt werden, um den DOS-Fehler zu erfragen. Ist zum Beispiel die angegebene Datei nicht vorhanden, liefern FOPEN() -1 und FERROR() den Wert 2, was besagt, da. die Datei nicht gefunden wurde. Eine komplette Liste der DOS-Fehlernummern ist bei der Funktion FERROR() angegeben.

Wurde die Datei erfolgreich ge.ffnet, wird die Dateikennung (filehandle) zur.ckgegeben. Diese Dateikennung ist wie ein Aliasname im Datenbanksystem zu verstehen, und wird dazu verwendet, auch durch andere Low-Level Dateifunktionen auf diese Datei zuzugreifen. Aus diesem Grund sollte die Dateikennung in einer Variablen abgelegt werden.

Warnung

Diese Funktion erlaubt den Zugriff auf Dateien auf Systemebene (low-level). Sie sollte nur mit .u.erster Vorsicht und genauen Kenntnissen des Betriebssystems angewendet werden.

Hinweise

Zugriff auf Dateien in anderen Verzeichnissen: FOPEN() ber.cksichtigt nicht die Einstellungen, die mit SET PATH oder SET DEFAULT vorgenommen worden sind. Ist im Parameter <cDateiName> kein Verzeichnis angegeben worden, sucht FOPEN() in den Verzeichnissen des DOS-Befehls PATH.

Beispiele

. Das folgende Beispiel .ffnet eine Datei f.r den Schreib- und Lesezugriff und gibt eine Fehlermeldung aus, wenn es nicht gelingt, die Datei zu .ffnen:

```
#include "FileIO.ch"
//
nKennung := FOPEN("Temp.txt", FO_READWRITE)
IF FERROR() <> 0
                                  ? "Datei kann nicht ge.ffnet werden. DOS-Error: ", FERROR()
ENDIF
```

Dateien

Bibliothek ist EXTEND.LIB, Headerdatei ist FileIO.ch

See Also:

FOUND() Ermittelt, ob eine vorhergehende Suchoperation erfolgreich war

---

Syntax

```
FOUND() -> lGefunden
```

R.ckgabe

FOUND() liefert true (.T.) zur.ck, wenn die letzte Suchoperation erfolgreich war, ansonsten wird false (.F.) zur.ckgegeben.

Anwendung

FOUND() ist eine Datenbankfunktion und wird in Verbindung mit Suchoperationen (z.B. FIND, LOCATE, CONTINUE, SEEK UND SET RELATION) benutzt, um festzustellen, ob die jeweilige Suchoperation erfolgreich war. Immer, wenn eine der Suchoperationen durchgef.hrt wird und der Begriff aufgefunden wurde, ist FOUND() auf true (.T.) gesetzt, andernfalls auf false (.F.).

Bei den Anweisungen LOCATE und CONTINUE ist die Suche erfolgreich, wenn ein Datensatz gefunden wird, der der Bedingung entspricht. Bei den Anweisungen FIND, SEEK oder SET RELATION ist der R.ckgabewert true (.T.), wenn der gesuchte Ausdruck mit dem entsprechenden Schl.ssel .bereinstimmt. Ist die Anweisung SET SOFTSEEK auf ON gestellt, wird der Satzzeiger auf den Datensatz positioniert, der dem gesuchten Ausdruck oder dem n.chst h.heren Wert entspricht. In diesem Fall liefert FOUND() true (.T.), wenn die Begriffe .bereinstimmen, ansonsten wird false (.F.) zur.ckgegeben.

Der R.ckgabewert von FOUND() wird so lange beibehalten, bis der Satzzeiger erneut bewegt wird. Wird der Satzzeiger nicht durch eine Suchoperation bewegt, liefert FOUND() immer false (.F.).

Jeder Arbeitsbereich besitzt einen eigenen FOUND()-Status. Das bedeutet, da. bei einer bestehenden Relation ein Aufruf der Funktion FOUND() in dem Arbeitsbereich, in den die Relation zeigt, nur dann als true (.T.) zur.ckgegeben wird, wenn die Suche der Relation erfolgreich war.

Als Voreinstellung arbeitet FOUND() im aktuellen Arbeitsbereich. Um auf einen anderen Arbeitsbereich zuzugreifen, verwenden Sie FOUND() mit einem vorangestellten Aliasbezeichner.

Beispiele

. Das folgende Beispiel zeigt FOUND() nach satzbewegenden Befehlen:

```
USE Verkauf INDEX Verkauf
? INDEXKEY(0)     // Resultat: VERKAEUFER
SEEK "Schmidt"
? FOUND()         // Resultat: .F.
SEEK "SCHMITZ"
? FOUND()         // Resultat: .T.
SKIP
? FOUND()         // Resultat: .F.
```

. Dieses Beispiel zeigt, wie FOUND() in einem nichtselektierten Arbeitsbereich angewendet werden kann:

```
USE Verkauf INDEX KundenNr NEW
USE Kunden INDEX Kunden NEW
SET RELATION TO KundenNr INTO Verkauf

SEEK "Schmidt"
? FOUND(), Verkauf->(FOUND())
```

. Das n.chste Beispiel bearbeitet alle Datens.tze mit dem Schl.sselausdruck "Schmidt":

```
USE Kunden INDEX Kunden NEW
SEEK "Schmidt"
```

```
DO WHILE FOUND()
__.
  .<Anweisungen>...
__.
  LOCATE REST WHILE Kunde = "Schmidt"
ENDDO
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

FREAD() Liest Zeichen aus einer Bin.r-Datei in eine Puffervariable

---

Syntax

FREAD(<nDateiKennung>, @<cPufferVar>, <nAnzahl>) -> nAnzahl

Parameter

<nDateiKennung> ist die Dateikennung, die von FOPEN(), FCREATE() oder von DOS zur.ckgegeben worden ist.

<cPufferVar> ist der Name einer Variablen vom Typ Zeichen, die als Puffer benutzt wird. Die Variable wird per Referenz (@) .bergeben und ihre L.nge mu. gr..er oder gleich <nAnzahl> sein.

<nAnzahl> ist die Anzahl der Zeichen, die in die Variable eingelesen werden sollen. Der R.ckgabewert einer erfolgreichen FREAD()-Operation mu. mit der Anzahl der Zeichen .bereinstimmen.

R.ckgabe

FREAD() gibt die Anzahl der erfolgreich eingelesenen Zeichen als ganzzahligen, numerischen Wert zur.ck. Ist der Wert kleiner als in <nAnzahl> angegeben, deutet das auf ein vorzeitiges Ende der Datei oder einen Lesefehler hin.

Anwendung

FREAD() ist eine Low-Level Dateifunktion und liest eine Anzahl von Zeichen aus einer Datei in eine existierende Zeichenkettenvariable. FREAD() beginnt mit dem Lesevorgang an der aktuellen Position des DOS-Dateizeigers und setzt den Zeiger gem.. der eingelesenen Anzahl von Zeichen neu. Es werden alle Zeichen, auch Kontroll-, Null- und Zeichen (oberhalb CHR(127) eingelesen.

FREAD() ist in einigen Eigenschaften vergleichbar mit den beiden Funktionen FREADSTR() und FSEEK. FREADSTR() liest eine Anzahl von Zeichen aus einer Datei, bis das Zeichen CHR(0) auftritt. FSEEK() bewegt den Dateizeiger, ohne eine Leseoperation durchzuf.hren.

Tritt w.hrend des Lesens ein Fehler auf, kann mit der Funktion FERROR() die Ursache ermittelt werden. Lesen Sie dazu auch unter FERROR() nach.

Warnung
Diese Funktion erlaubt den Zugriff auf Dateien auf Systemebene. Sie sollte nur mit .u.erster Vorsicht und genauen Kenntnissen des Betriebssystems angewendet werden.

Beispiele

Im folgenden Beispiel werden aus einer Datei 128 Zeichen in eine Puffervariable eingelesen:

```
#define F_BLOCK 128
//
cPuffer := SPACE(F_BLOCK)
nKennung := FOPEN("Temp.txt")
//
IF FERROR() <> 0
  ? "Datei.ffnungsfehler : ", FERROR()
ELSE
  IF FREAD(nKennung, @cPuffer, F_BLOCK) <> F_BLOCK
  ? "Lesefehler von Datei Temp.txt "
  ENDIF
  FCLOSE(nKennung)
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

FREADSTR() Liest eine Zeichenkette aus einer Bin.r-Datei

Syntax

FREADSTR(<cDateiKennung>, <nAnzahl>) -> cZeichenkette

Parameter

<nDateiKennung> ist die Dateikennung, die von FOPEN(), FCREATE() oder von DOS zur.ckgegeben worden ist.

<nAnzahl> ist die Anzahl der Zeichen, die, beginnend ab der aktuellen Position des DOS-Dateizeigers, eingelesen werden sollen.

R.ckgabe

FREADSTR() gibt die eingelesenen Zeichen als Zeichenkette zur.ck. Die maximale Gr..e der Zeichenkette darf dabei 65.520 (64 K) Bytes betragen. Tritt ein Fehler auf oder wird das Dateiende erreicht, wird eine leere Zeichenkette ("") zur.ckgegeben.

Anwendung

FREADSTR() ist eine Low-Level Dateifunktion und liest Zeichen aus einer ge.ffneten Datei ein, beginnend an der aktuellen Position des DOS-Dateizeigers. Es werden so viele Zeichen eingelesen, bis <nAnzahl> oder das Null-Zeichen (CHR(0)) erreicht ist. Es werden alle m.glichen Zeichencodes (einschlie.lich der Kontroll-Zeichen) eingelesen, mit Ausnahme des Zeichens CHR(0). Der Dateizeiger wird um die Anzahl der gelesenen Zeichen weiterbewegt. Ist <nAnzahl> gr..er als die Anzahl der Zeichen von der aktuellen Position des Zeigers bis zum Dateiende, wird der Zeiger auf das letzte Byte der Datei gesetzt.

Beispiele

Das folgende Beispiel zeigt die ASCII-Werte der ersten 16 Zeichen einer Datei an:

```
#include "FileIO.ch"
nKennung := FOPEN("Neu.txt", FC_NORMAL)
IF FERROR() <> 0
  ? "Datei.ffnungsfehler : ", FERROR()
  BREAK
ELSE
  cPuffer := FREADSTR(nKennung, 16)
  ? cPuffer
  FCLOSE(nKennung)
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

FRENAME() Wechselt den Namen einer Datei

Syntax

FRENAME(<cDateiNameAlt>, <cDateiNameNeu>) -> nAusgef.hrt

Parameter

<cDateiNameAlt> ist der umzubenennende Dateiname, einschlie.lich Dateinamenserweiterung. Ein Laufwerksbezeichner und/oder Verzeichnisname kann als Teil des Dateinamens enthalten sein.

<cDateiNameNeu> ist der neue Dateiname, einschlie.lich Dateinamenserweiterung. Ein Laufwerksbezeichner und/oder Verzeichnisname kann als Teil des Dateinamens enthalten sein.

R.ckgabe

FRENAME() liefert -1, wenn die Funktion nicht durchgef.hrt werden konnte, andernfalls 0. Wurde die Funktion nicht durchgef.hrt, kann mit FERROR() die Fehlerursache festgestellt werden.

Anwendung

FRENAME() ist eine Dateifunktion, die den Namen einer vorhandenen Datei gegen einen neuen Namen wechselt, und identisch mit dem Befehl RENAME. Wird FRENAME() aufgerufen, so wird <cDateiNameAlt> nur dann umbenannt, wenn sich die Datei im aktuellen Verzeichnis befindet oder die Pfadangabe ausdr.cklich im Dateinamen enthalten ist. Die Einstellungen durch SET PATH und SET DEFAULT werden von FRENAME() nicht ber.cksichtigt. Sind die Pfadangaben in <cDateiNameAlt> und <cDateiNameNeu> unterschiedlich, so wechselt die Datei in das Zielverzeichnis. Falls <cDateiNameNeu> bereits existiert oder ge.ffnet ist, erfolgt keine Umbenennung der Datei und FRENAME() liefert -1 als Fehler. Die genaue Ursache des Fehlers kann .ber FERROR() ermittelt werden.

Warnung

Dateien m.ssen, bevor sie umbenannt werden sollen, geschlossen sein. Der Versuch, eine ge.ffnete Datei umzubenennen, liefert unvorhersehbare Resultate. Wenn eine Datenbank-Datei (.dbf) umbenannt wird, die Memofelder enth.lt, so ist ebenfalls die entsprechende Memo-Datei (.dbt) umzubenennen.

Beispiele

Das folgende Beispiel demonstriert, wie eine Datei umbenannt werden kann:

```
IF FRENAME("DateiAlt.txt", "DateiNeu.txt) = -1
  ? "Dateifehler : ", FERROR()
  BREAK
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

FSEEK() Setzt den Dateizeiger einer Bin.r-Datei auf eine neue Position

---

Syntax

FSEEK(<nDateiKennung>, <nOffset>, [<nUrsprung>]) -> nNeuePosition

Parameter

<nDateiKennung> ist die Dateikennung, die von FOPEN(), FCREATE() oder von DOS zur.ckgegeben worden ist.

<nOffset> ist die Anzahl der Bytes, um die der Dateizeiger, beginnend bei <nUrsprung>, verschoben werden soll. Die Anzahl der Bytes kann sowohl positive als auch negative Werte annehmen, entsprechend der Richtung, in die der Zeiger bewegt werden soll.

<nUrsprung> definiert die Startposition, ab der die Verschiebung des Dateizeigers um <nOffset> beginnen soll. Der Vorgabewert betr.gt 0 und steht f.r den Dateianfang. Wenn <nUrsprung> das Dateiende repr.sentiert, mu. <nOffset> Null oder einen negativen Wert enthalten.

Tabelle 5-19: FSEEK() Methoden z. Bewegen des Dateizeigers

Ursprung	FileIO.ch	Beschreibung
0	FS_SET	Beginn beim Dateianfang
1	FS_RELATIVE	Beginn bei der aktuellen Position
2	FS_END	Beginn beim Dateiende

R.ckgabe

FSEEK() gibt die neue Position des Dateizeigers relativ zum Dateianfang (Position 0) als numerischen Wert zur.ck. Dieser Wert besitzt keinen Bezug zur alten Position des Dateizeigers.

Anwendung

FSEEK() ist eine Low-Level Dateifunktion und bewegt den Dateizeiger vorw.rts oder r.ckw.rts innerhalb einer ge.ffneten Datei. Zeichen werden dabei nicht gelesen. Die Position, zu welcher der Dateizeiger bewegt wird, ist durch <nOffset> und <nUrsprung> bestimmt. Der R.ckgabewert gibt die neue Position des Dateizeigers an. Unabh.ngig von den angegebenen Parametern kann der Dateizeiger weder .ber den Dateibeginn, noch .ber das Dateiende hinaus bewegt werden.

Warnung

Diese Funktion erlaubt den Zugriff auf Dateien auf Systemebene. Sie sollte nur mit
.u.erster Vorsicht und genauen Kenntnissen des Betriebssystems angewendet werden.

Beispiele

. Das folgende Beispiel ermittelt die L.nge einer Datei, indem FSEEK() den Dateizeiger vom Dateiende um Null Bytes bewegt. Danach wird der Dateizeiger auf den Dateianfang gesetzt:

```
#include "FileIO.ch"
//
// .ffnen der Datei im Lesezugriff
IF (nKennung := FOPEN("Temp.txt")) >= 0
  // Ermitteln der Dateil.nge
  nLaenge := FSEEK(nKennung, 0, FS_END)
  // Setzen des Zeigers auf den Dateianfang
  FSEEK(nKennung, 0)
ELSE
  ? "Fehler beim .ffnen der Datei, Fehler: ", FERROR()
ENDIF
```

. Um den Dateizeiger auf das letzte Byte einer Datei zu bewegen, benutzen Sie die folgende Pseudo-Funktion:

```
# define D_ENDE(nKennung)  (FSEEK(nKennung, 0, 2))
```

. Um den Dateizeiger auf das erste Byte einer Datei zu bewegen, benutzen Sie die folgende Pseudo-Funktion:

```
# define D_ANFANG(nKennung)  (FSEEK(nKennung, 0))
```

. Um die aktuelle Position des Dateizeigers zu ermitteln, benutzen Sie die folgende Pseudo-Funktion:

```
# define D_AKTUELL(nKennung)  (FSEEK(nKennung, 0, 1))
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist FileIO.ch

See Also:

FWRITE() Schreibt eine Zeichenkette in eine ge.ffnete Bin.r-Datei

---

Syntax

FWRITE(<nDateiKennung>, <cPuffer>, [<nAnzahl>]) -> nAnzahl

Parameter

<nDateiKennung> ist die Dateikennung, die von FOPEN(), FCREATE() oder von DOS zur.ckgegeben worden ist.

<cPuffer> ist die Zeichenkette, die in die angegebene Datei geschrieben werden soll.

<nAnzahl> ist die Anzahl der Zeichen, die beginnend ab der Position des Dateizeigers aus <cPuffer> in die Datei geschrieben werden soll. Wird dieser Wert nicht angegeben, schreibt FWRITE() alle Zeichen aus <cPuffer> in die Datei.

R.ckgabe

FWRITE() gibt die Anzahl der tats.chlich geschriebenen Zeichen als numerischen Wert zur.ck. Ist der Wert gleich <nAnzahl>, war die Operation erfolgreich. Ist der Wert ungleich <nAnzahl> oder 0, ist auf der Diskette oder Festplatte nicht ausreichend Speicherplatz vorhanden oder es trat ein anderer Fehler auf. Die genaue Ursache des Fehlers kann mit FERROR() ermittelt werden.

Anwendung

FWRITE() ist eine Low-Level Dateifunktion und schreibt eine Anzahl von Zeichen aus einem Puffer in eine zuvor ge.ffnete Datei. Dabei kann entweder der gesamte Puffer oder nur ein Teil des Puffers geschrieben werden. Der Schreibvorgang beginnt ab der aktuellen Position des Dateizeigers. Die Anzahl der geschriebenen Zeichen wird als Ergebnis zur.ckgegeben.

Sollte bei dem Schreibvorgang ein Fehler auftreten, kann die genaue Ursache mit der Funktion FERROR() ermittelt werden.

Warnung
Diese Funktion erlaubt den Zugriff auf Dateien auf Systemebene. Sie sollte nur mit .u.erster Vorsicht und genauen Kenntnissen des Betriebssystems angewendet werden.

Beispiele

. In diesem Beispiel wird der Inhalt einer Datei in eine andere kopiert:

```
#include "FileIO.ch"
#define F_BLOCK 512
//
cPuffer     := SPACE(F_BLOCK)
nQuelldatei := FOPEN("Temp.txt", FO_READ)
nZieldatei  := FCREATE("Neu.txt", FC_WRITE)
lFertig     := .F.
//
DO WHILE .NOT. lFertig
  nGelesen  := FREAD(nQuelldatei, @cPuffer, F_BLOCK)
  IF FWRITE(nZieldatei, cPuffer, nGelesen) < nGelesen
    ? "Schreibfehler: ", FERROR()
    lFertig = .T.
  ELSE
    lFertig   := (nGelesen == 0)
  ENDIF
ENDDO
//
FCLOSE(nQuelldatei)
FCLOSE(nZieldatei)
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

GETACTIVE() Liefert das aktuelle, aktive GET-Objekt

Syntax

GETACTIVE() -> objGET

R.ckgabe

GETACTIVE liefert das aktuelle, aktive Get-Objekt innerhalb eines READ's. Ist READ w.hrend des Aufrufes der Funktion GETACTIVT nicht aktiv, liefert die Funktion NIL.

Anwendung

GETACTIVE() ist eine Environment-Funktion, die den Zugriff auf ein aktives GET w.hrend der Abarbeitung eines READ's erlaubt. Das aktive GET-Objekt ist das GET, das w.hrend des READ's hell dargestellt ist.

Beispiele

Das folgende Beispiel gebraucht die WHEN-Option eines GET-Befehls, um in eine Spezial-Funktion zu verzweigen. Innerhalb dieser Funktion wird die GETACTIVE()-Funktion benutzt, um das aktive Get-Objekt zu ermitteln:

```
@ 10, 10 GET x
@ 12, 10 GET y WHEN MyReader()
@ 14, 10 GET z

FUNCTION MyReader
LOCAL objGet
  objGet := GETACTIVE()
  BarCodeRead( objGet )
  RETURN(.f.)
```

Dateien

Bibliothek ist CLIPPER.LIB, Quelldatei ist GETSYS.PRG

See Also:

GETENV() Ermittelt den Inhalt einer DOS-Umgebungsvariablen

Syntax

GETENV(<cUmgebungsVariable>) -> cZeichenkette

Parameter

<cUmgebungsVariable> ist der Name der DOS-Umgebungsvariablen. Die Angabe in diesem Parameter kann Gro.- und Kleinbuchstaben enthalten; der Parameter ist nicht Case-Sensitiv.

R.ckgabe

GETENV() gibt den Inhalt einer DOS-Umgebungsvariablen als Zeichenkette zur.ck. Wird die Umgebungsvariable nicht gefunden, wird eine leere Zeichenkette ("") zur.ckgegeben.

Anwendung

GETENV() ist eine Environment-Funktion, welche erlaubt, Konfigurationsinformationen aus der DOS-Umgebung in die eigene Applikation zu .bernehmen. Ein typisches Anwendungsbeispiel ist, wenn Informationen .ber Verzeichnis- und Laufwerksangaben aus einer DOS-Variablen entnommen werden, aus der ersichtlich ist, wo sich die verwendeten Daten befinden (z.B. Datenbanken, Indexdateien, Label- oder Reportdateien). Auch in der Netzwerkumgebung ist diese Funktion sehr hilfreich.

Hinweise

. " des DOS SET-Kommandos ein Leerzeichen angegeben ist und entfernen Sie dieses gegebenenfalls.

. Kompatibilit.t: In fr.heren Versionen von Clipper hie. diese Funktion GETE(). Diese abgewandelte Form von GETENV() steht weiterhin zur Verf.gung.

Beispiele

. Dieses Beispiel holt die aktuelle DOS-Verzeichnisliste und setzt den Pfad in die Clipper-PATH Liste:

```
cPfad := GETENV("PATH")
SET PATH TO (cPfad)
```

. Das n.chste Beispiel zeigt Ihnen, wie Sie Umgebungsvariablen benutzen k.nnen, um verschiedene Verzeichnisangaben f.r die von Ihrem Programm ben.tigten Dateien zu implementieren. Dazu belegen Sie zun.chst folgende Umgebungsvariablen mit dem DOS SET-Kommando:

```
SET DBF_PFAD=<Verzeichnis Ihrer Datenbanken>
SET NTX_PFAD=<Verzeichnis Ihrer Indexdateien>
SET RPT_PFAD=<Verzeichnis Ihrer Reportformatdateien>
```

. In dem konfigurierenden Teil Ihres Programms weisen Sie dann die Werte der Umgebungsvariablen Speichervariablen zu und verwenden sie wie folgt, um auf die ben.tigten Dateien zu zugreifen:

```
cDbfDir := GETENV("DBF_PFAD")
USE (cDbfDir + "Rechnung")
```

Dateien

HARDCR() Ersetzt in einer Zeichenkette alle automatischen Zeilenumbr.che gegen harte Zeilenumbr.che

---

Syntax

HARDCR(<cZeichenkette>) -> cKonvertiert

Parameter

<cZeichenkette> ist die Zeichenkette oder das Memofeld, das zu konvertieren ist.

R.ckgabe

HARDCR() liefert eine Zeichenkette mit einer L.nge von maximal 65.520 Zeichen.

Anwendung

HARDCR() ist eine Memofunktion, die alle automatischen Zeilenumbr.che (CHR(141)) gegen harte Zeilenumbr.che (CHR(13)) ersetzt. HARDCR() wird benutzt, um Zeichenkettenvariablen oder Memofelder, in denen eine weiche Zeilenschaltung enthalten ist, mit Ausgabe-Befehlen (einschlie.lich REPORT FORM und REPORT LABEL) anzuzeigen. In Clipper wird eine solche automatische Konvertierung nicht vorgenommen. Automatische Zeilenschaltungen werden von MEMOEDIT() eingesetzt, wenn ein Wort nicht mehr in eine Zeile pa.t.

Hinweise

HARDCR() in Report- und Labelformaten: Tritt HARDCR() nur als Bestandteil eines Ausdrucks im Report- oder Listenformat auf und wird ansonsten in Ihrem Programm nicht verwendet, m.ssen Sie HARDCR() als EXTERNAL deklarieren, um sicherzustellen, da. die Funktion zusammen mit Ihrer Anwendung gelinkt wird.

Beispiele

Das Beispiel zeigt die Darstellung eines Memofeldes, das .ber MEMOEDIT() mit automatischen Zeilenumbr.chen formatiert wurde:

```
USE Kunden NEW
? HARDCR(Kunden->Notizen)
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

HEADER() Liefert die L.nge des Dateikopfes der aktuellen Datenbank

---

Syntax

HEADER() -> nByte

R.ckgabe

HEADER() liefert die Anzahl der Byte im Kopf der aktuellen Datenbankdatei als numerischen, ganzzahligen Wert.

Anwendung

HEADER() ist eine Datenbankfunktion, welche mit den Funktionen LASTREC(), RECSIZE() und DISKSPACE benutzt wird, um Prozeduren zum Backup von Datenbanken zu erstellen.

Als Voreinstellung arbeitet HEADER() im aktuellen Arbeitsbereich. Diese Funktion kann auch auf nichtselektierte Arbeitsbereiche angewendet werden, indem der Funktion ein Aliasbezeichner vorangestellt wird.

Beispiele

. Dieses Beispiel ermittelt die Kopfgr..e der Datenbank Verkauf:

```
USE Verkauf NEW
? HEADER()  // Resultat: 258
```

. Dieses Beispiel definiert eine Pseudo-Funktion DbfSize(), die HEADER(), RECSIZE() und LASTREC() verwendet, um die Gr..e einer Datenbank in Byte zu ermitteln:

```
#define DbfSize() ((RECSIZE() * LASTREC())  + HEADER() + 1)
```

. Diese Funktion kann sp.ter wie jede andere Funktion benutzt werden:

```
USE Verkauf NEW
USE Kunden NEW

? DbfSize()
? Verkauf->(DbfSize())
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

I2BIN() Konvertiert einen numerischen Wert in einen Bin.rwert

---

Syntax

I2BIN(<nInteger>) -> cBin.rInteger

Parameter

<nInteger> ist der zu konvertierende numerische Wert. Kommastellen werden abgeschnitten.

R.ckgabe

I2BIN() liefert eine zwei Byte lange Zeichenfolge, di einen 16-Bit Bin.rwert darstellt.

Anwendung

I2BIN() ist eine Low-Level Dateifunktion, die einen numerischen Wert in eine zwei Byte lange Zeichenkette umwandelt (h.herwertiges Byte zuerst). I2BIN() wird in Verbindung mit FWRITE() benutzt, um Standard-Bin.rwerte schreiben zu k.nnen. Die inverse Funktion zu I2BIN() ist BIN2I().

Beispiele

Dieses Beispiel .ffnet eine Datenbank mit Hilfe einer Low-Level Dateifunktion und ver.ndert das Datum der letzten .nderung:

```
#include "FileIO.ch"
//
nKennung  := FOPEN("Verkauf.dbf", FO_READWRITE )
//
// Neues .nderungsdatum konvertieren
//
nJahr     := I2BIN(90)
nMonat    := I2BIN(12)
nTag.:= I2BIN(15)
//
// Dateizeiger setzen
FSEEK(nKennung, 1, FS_SET)
//
// Das neue .nderungsdatum in Datei schreiben,
// jeweils nur das niederwertige Byte
FWRITE(nKennung, nJahr, 1)
FWRITE(nKennung, nMonat, 1)
FWRITE(nKennung, nTag, 1)
//
FCLOSE(nKennung)
```

Dateien

Bibliothek ist CLIPPER.LIB, Quelldatei ist EXAMPLEA.ASM

See Also:

IF() Liefert das Ergebnis eines Ausdrucks in Abh.ngigkeit von einer Bedingung

---

Syntax

[I]IF(<lBedingung>, <expTrue>, <expFalse>) -> Wert

Parameter

<lBedingung> ist ein logischer Ausdruck, der ausgewertet werden soll.

<expTrue> ist ein Wert beliebigen Typs, der zur.ckgegeben wird, wenn <lBedingung> den Wert true (.T.) ergibt.

<expFalse> ist ein Wert beliebigen Typs, der zur.ckgegeben wird, wenn <lBedingung> den Wert false (.F.) ergibt. Dieser Parameter mu. nicht vom gleichen Datentyp wie <expTrue> sein.

R.ckgabe

IF() gibt als Wert das Ergebnis von <expTrue> zur.ck, wenn die Bedingung <lBedingung> true (.T.) ergibt. Ist die Bedingung false (.F.), wird das Ergebnis von <expFalse> zur.ckgegeben. Da die beiden Ausdr.cke von verschiedenen Datentypen sein d.rfen, kann auch der R.ckgabewert verschiedenen Datentyps sein.

Anwendung

IF() ist eine logische Konvertierungsfunktion. Sie stellt einen Mechanismus zur Verf.gung, mit dem eine Bedingung innerhalb eines Ausdrucks ermittelt werden kann. Auf Grund dieser F.higkeit k.nnen Sie zum Beispiel einen logischen Datentyp in jeden anderen Typ umwandeln.

Beispiele

. Das folgende Beispiel zeigt die Konvertierung eines logischen Datentyps in einen numerischen:

```
lBezahlt := .T.
? IF(lBezahlt, 1, 0)  // Resultat: 1
```

. Im folgenden wird die Ausgabe einer logischen Variable in Abh.ngigkeit von ihrem Wert formatiert:

```
@ ROW() + 1, 25 SAY IF(lBezahlt, SPACE(10), "Mahnung schreiben")
```

. In Formaten k.nnen Sie in Abh.ngigkeit von einem logischen Feld Ausgaben an verschiedenen Positionen vornehmen:

```
@ ROW() + 1, IF(lBezahlt, 10, 12) SAY "X"
```

. Sie k.nnen IF() ebenfalls dazu benutzen, um in einem Reportformat leere Zeilen zu erzeugen:

```
IF( EMPTY(Firma), CHR(255), Firma)
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

INDEXEXT() Ermittelt die Dateinamenserweiterung der Indexdatei anhand des verwendeten Datenbanktreibers

---

Syntax

INDEXEXT() -> cDateinamenserweiterung

R.ckgabe

Sofern Sie keinen anderen Datenbanktreiber angegeben haben, liefert INDEXEXT() ".NTX". Daran k.nnen Sie feststellen, da. der standardgem..e Clipper-Treiber aktiv ist. Ist der sprachkompatible Treiber NDX.OBJ zur Applikation hinzugelinkt worden, liefert die Funktion ".NDX".

Anwendung

INDEXEXT() ermittelt die Dateinamenserweiterung der Indexdatei, um festzustellen, welcher Datenbanktreiber aktuell hinzugelinkt wurde.

Beispiele

In diesem Beispiel wird festgestellt, ob der korrekte Index zu der Datenbank Kunden existiert, und gegebenenfalls neu erzeugt:

```
USE verkauf NEW
IF .NOT. FILE( "Kunden" + INDEXEXT())
INDEX ON Kunden TO Kunden
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

INDEXKEY() Ermittelt den Schl.sselausdruck eines Indexes

Syntax

INDEXKEY(<nOrder>) -> cIndexSchl.ssel

Parameter

<nOrder> ist die Ordnungsnummer des in der Indexliste angegebenen Indexes. Diese gilt f.r die zuletzt .ber USE...INDEX oder SET INDEX TO ge.ffneteten Indexdateien des aktuellen Arbeitsbereiches. Null als Parameter liefert dabei den Ausdruck des f.hrenden Indexes (Hauptindex), unabh.ngig von der Position in der Indexliste, zur.ck.

R.ckgabe

INDEXKEY() liefert den Schl.sselausdruck des .ber <nOrder> spezifizierten Indexes in Form einer Zeichenkette. Ist kein zu <nOrder> korrespondierender Index vorhanden, liefert INDEXKEY() eine leere Zeichenkette ("").

Anwendung

INDEXKEY() ist eine Datenbankfunktion. Sie ermittelt den Schl.sselausdruck eines spezifizierten Indexes des aktuellen Arbeitsbereiches und gibt diesen als Zeichenkette zur.ck. Um einen Indexausdruck zu berechnen, wird INDEXKEY() in einem Makroausdruck verwendet:

```
&(INDEXKEY(<nOrder>)).
```

INDEXKEY() dient einer Reihe von Anwendungen, von denen hier zwei hervorgehoben werden sollen. Wenn der Befehl TOTAL benutzt wird, kann INDEXKEY() anstelle eines Schl.sselausdruckes im Quellcode angegeben werden. Eine andere Anwendungsm.glichkeit besteht innerhalb der Funktion DBEDIT(). Hierbei kann .ber INDEXKEY() festgestellt werden, ob nach dem Editieren eines Datensatzes der Bildschirm neu aufgebaut werden mu., oder nicht. Normalerweise ist dies nur erforderlich, wenn sich der Wert des Schl.sselausdrucks des f.hrenden Indexes .ndert. Beispiele f.r beide F.lle werden weiter unten gezeigt.

Als Voreinstellung arbeitet INDEXKEY() im aktuellen Arbeitsbereich. Durch Voranstellen eines Aliasbezeichers kann die Funktion auch auf nichtselektierte Arbeitsbereiche angewendet werden.

Beispiele

. Dieses Beispiel zeigt die Anwendung von INDEXKEY(), um den Indexausdruck des aktuellen Arbeitsbereiches zu ermitteln:

```
#define ORD_NORMAL  0
#define ORD_NAME  1
#define ORD_NUMMER  2
//
USE Kunden INDEX Name, Nummer NEW
SET ORDER TO ORD_NUMMER
? INDEXKEY(ORD_NAME)       // Resultat: Name
? INDEXKEY(ORD_NUMMER)     // Resultat: Nummer
? INDEXKEY(ORD_NORMAL)     // Resultat: Nummer
```

. Das n.chste Beispiel zeigt Ihnen, wie Sie auf den Indexausdruck eines nichtselektierten Arbeitsbereiches zugreifen:

```
USE Kunden INDEX Name, Nummer NEW
USE Verkauf INDEX Verkaeuf NEW
? INDEXKEY(0), Kunden->(INDEXKEY(0))
```

. Dieses Beispiel verwendet INDEXKEY() als Makroausdruck bei einer TOTAL-Anweisung:

```
USE Verkauf INDEX Verkaeuf NEW
TOTAL ON &(INDEXKEY(0)) FIELDS Betrag TO Summe
```

. Das letzte Beispiel veranschaulicht, wie mit Hilfe von INDEXKEY()
innerhalb von DBEDIT() festgestellt werden kann, ob der Bildschirm
aktualisiert werden mu., wenn der Benutzer einen Datensatz .ndert.
Grunds.tzlich ist dies nur erforderlich, wenn ein Datenfeld ge.ndert
wird, das Bestandteil vom Schl.sselausdruck des f.hrenden Indexes ist.
FeldEdit() ist hier eine Anwenderfunktion (UDF), die von DBEDIT()
aufgerufen wird, wenn der Benutzer eine Taste bet.tigt.

```
#include "Dbedit.ch"
#define ORD_NORMAL 0
//
FUNKTION FeldEdit()
  LOCAL IndexWert, nReturn := DB_CONT
  // Sichern des aktuellen Indexwertes
  IndexWert := &(INDEXKEY(ORD_NORMAL))
  .
  .<GET-Anweisungen, f.r das aktuelle Feld>
  .
  // Bildschirm aktualisieren, wenn sich der Indexwert .ndert
  IF IndexWert <> &(INDEXKEY(ORD_NORMAL))
    nReturn := DB_REFRESH
  ENDIF
RETURN nReturn
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

INDEXORD() Liefert die Ordnungsnummer des f.hrenden Indexes

Syntax

INDEXORD() -> nOrder

R.ckgabe

INDEXORD() liefert einen ganzzahligen, numerischen Wert. Dieser Wert gibt die Position des f.hrenden Indexes in der Reihe der zuletzt ge.ffneten Indexdateien des aktuellen Arbeitsbereiches an. Null als R.ckgabewert bedeutet, da. kein Index als f.hrender Index benutzt wird, und die Reihenfolge der Datensatznummern gilt.

Anwendung

INDEXORD() ist eine Datenbankfunktion, welche die Ordnungsnummer des f.hrenden Indexes in der Reihe der zuletzt .ber USE...INDEX oder SET INDEX TO ge.ffneten Indexdateien im aktuellen Arbeitsbereich ermittelt. Diese Funktion ist oft sehr n.tzlich, wenn der f.hrende Index zu Wiederherstellungszwecken gesichert werden soll.

Als Voreinstellung arbeitet INDEXORD() im aktuellen Arbeitsbereich. Durch Voranstellen eines Aliasbezeichners kann INDEXORD() auch auf nichtselektierte Arbeitsbereiche angewendet werden.

Beispiele

. Dieses Beispiel zeigt die Anwendung von INDEXORD(), um eine Sortierreihenfolge zu einem sp.teren Zeitpunkt wiederherzustellen:

```
USE Kunden INDEX Name, Nummer NEW
nOrder := INDEXORD()       // Resultat: 1
SET ORDER TO 2
? INDEXORD()               // Resultat: 2
SET ORDER TO nOrder
? INDEXORD()               // Resultat: 1
```

. Das folgende Beispiel benutzt einen Aliasbezeichner, um auf einen anderen Arbeitsbereich zuzugreifen:

```
USE Verkauf INDEX Verk, KundenNr NEW
USE Kunden INDEX Name, Nummer NEW
? Verkauf->(INDEXORD())    // Resultat: 1
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

INKEY() Holt ein Zeichen aus dem Tastaturpuffer

Syntax

INKEY([<nSekunden>]) -> nInkeyCode

Parameter

<nSekunden> gibt die Zeit in Sekunden an, die INKEY() auf eine Eingabe von der Tastatur wartet, wenn kein Tastencode im Tastaturpuffer hinterlegt wurde. Der Wert kann in Schritten von 10tel Sekunden angegeben werden. Eine Null als Parameter veranla.t INKEY() so lange zu warten, bis der Anwender eine Taste bet.tigt. Wird kein Parameter angegeben, holt INKEY() lediglich den als n.chstes anstehenden Tastencode und wartet nicht auf einen Tastendruck, wenn der Tastaturpuffer leer ist.

R.ckgabe

INKEY() liefert einen ganzzahligen, numerischen Wert zwischen -39 und 386 zur Identifizierung der bet.tigten Taste. Ist w.hrend der Wartezeit keine Taste bet.tigt worden, liefert die Funktion den Wert Null. INKEY() liefert Werte f.r alle ASCII-Zeichen-, Funktions-, Alt-Funktions-, Strg-Funktions-, Alt-Ziffern-, Strg-Ziffern- und Cursortasten.

Anwendung

INKEY() ist eine Tastaturfunktion und holt den n.chsten anstehenden Code aus dem Tastaturpuffer, welcher einer Taste zugeordnet ist. Dieser Wert bleibt intern erhalten und kann .ber LASTKEY() abgefragt werden. Wird der Parameter <nSekunden> angegeben und stehen keine Tastencodes im Tastaturpuffer, wartet INKEY() so lange, bis entweder durch Bet.tigung einer Taste ein neuer Tastencode in den Tastaturpuffer gestellt wird, oder die Wartezeit abgelaufen ist. Die Wartezeit, auf die sich INKEY() bezieht, wird von der internen Systemuhr .bernommen, so da. der Prozessortakt darauf keinen Einflu. hat. Ist <nSekunden> mit Null angegeben, wartet INKEY() so lange, bis ein neuer Tastenwert in den Tastaturpuffer gestellt wird.

Beachten Sie, da. INKEY() kein Wartezustand ist und somit Anweisungen .ber SET KEY TO nicht aktiv sind.

INKEY() ist .hnlich wie NEXTKEY(). Im Gegensatz zu INKEY() liest NEXTKEY() lediglich den n.chsten Code aus dem Tastaturpuffer, ohne ihn abzuholen. Dies ist sehr hilfreich, wenn lediglich getestet werden soll, ob und was f.r eine Taste ansteht, ohne den Tastencode zu verarbeiten.

INKEY() ist im Clipper-System eine Basisfunktion zum Abholen eines Tastenwertes aus dem Tastaturpuffer. Die Funktion wird zur Programmwahl mittels Tastendruck ben.tigt oder zur Unterbrechung eines Programms, bis eine Taste bet.tigt wird. Sie k.nnen INKEY() aber auch benutzen, um auf Tastendruck eine Anweisung wie LIST, LABEL FORM und REPORT FORM, oder auch Programmschleifen wie DO WHILE zu unterbrechen (siehe Beispiele).

Eine komplette Liste der INKEY()-Codes und in Inkey.ch definierten Konstanten enth.lt der Anhang C des Referenzhandbuches.

Beispiele

. Das folgende Beispiel zeigt das Zeichen [Esc] und dessen Code, sobald eine Taste gedr.ckt worden ist:

```
#include "Inkey.ch"
//
LOCAL nInkeyCode := 0
DO WHILE LASTKEY() <> K_ESC
  ? "Dr.cken Sie eine Taste ..."
  nInkeyCode := INKEY(0)
  ?? "Zeichen: ", CHR(nInkeyCode),;
       "Code:", TRANSFORM(nInkeyCode,"999")
ENDDO
```

. Dies Beispiel zeigt, wie INKEY() in einer WHILE-Bedingung verwendet
werden kann, um ein Kommando mit der Taste abzubrechen:

```
#include "Inkey.ch"
//
USE Verkauf INDEX Verkauf NEW
REPORT FORM Monat FOR MONTH(VerkDatum) = MONTH(DATE());
       WHILE INKEY() <> K_ESC
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

INT() Konvertiert einen numerischen Wert in eine Ganzzahl

Syntax

INT(<nExp>) -> nGanzzahl

Parameter

<nExp> ist der zu konvertierende, numerische Ausdruck.

R.ckgabe

INT() gibt einen ganzzahligen Wert (integer) zur.ck.

Anwendung

INT() ist eine numerische Funktion, die einen numerischen Wert in einen ganzzahligen Wert konvertiert, indem die Nachkommastellen abgeschnitten werden (nicht gerundet!). INT() wird immer dann benutzt, wenn Sie die Nachkommastellen einer Zahl nicht ben.tigen.

Beispiele

Das Beispiel zeigt die Ergebnisse der verschiedenen Aufrufe von INT():

```
? INT(100.00)        // Resultat: 100
? INT(0.5)           // Resultat: 0
? INT(-100.75)       // Resultat: -100
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

ISALPHA() Buchstabe>mittelt, ob das erste Zeichen in einer Zeichenkette ein Buchstabe ist

---

Syntax

ISALPHA(<cZeichenkette>) -> lAlpha

Parameter

<cZeichenkette> ist die zu untersuchende Zeichenkette.

R.ckgabe

ISALPHA() liefert true (.T.), wenn das erste Zeichen der Zeichenkette ein Buchstabe ist, ansonsten wird false (.F.) zur.ckgegeben.

Anwendung

ISALPHA() ist eine Zeichenfunktion, die feststellt, ob das erste Zeichen der .bergebenen Zeichenkette ein Buchstabe ist. Der Buchstabe darf dabei gro. oder klein geschrieben sein und liegt im Bereich von(a-Z, .-.). Falls die angegebene Zeichenkette mit einer Ziffer oder einem anderen Zeichen beginnt, liefert ISALPHA() den Wert false (.F.).

Beispiele

Das Beispiel zeigt die verschiedenen Resultate von ISALPHA():

```
? ISALPHA("AbcDe")  // Resultat: .T.
? ISALPHA("aBcDe")  // Resultat: .T.
? ISALPHA("1BCdE")  // Resultat: .F.
? ISALPHA(".FRED")  // Resultat: .F.
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

ISCOLOR() Ermittelt, ob ein Farbbildschirm angeschlossen ist

Syntax

ISCOLOR() | ISCOLOUR() -> lFarbe

R.ckgabe

ISCOLOR() gibt true (.T.) zur.ck, wenn ein Farb-Bildschirmadapter angeschlossen ist, ansonsten wird false (.F.) zur.ckgegeben.

Anwendung

ISCOLOR() ist eine Bildschirmfunktion, die Ihnen die M.glichkeit bietet, in Abh.ngigkeit von der verwendeten Bildschirmkarte (farbig oder monochrom) zu entscheiden, welche Farbattribute Sie in Ihrem Programm verwenden. Beachten Sie, da. manche monochromen Adapterkarten mit Grafikm.glichkeit ebenfalls true (.T.) zur.ckgeben.

Beispiele

Das folgende Beispiel installiert zur Laufzeit eines Programms Variablen mit entsprechenden Farbwerten:

```
IF ISCOLOR()
   cRahmen    := "BG+/B, W/N"
   cHell      := "+W/B, N/W"
   cNormal    := "W/B, N/W"
   cInvers    := "B/W, W/N"
ELSE
   cRahmen    := "W"
   cHell      := "+W/N, N/+W"
   cNormal    := "W/N, N/W"
   cInvers    := "N/W, W/N"
ENDIF
SETCOLOR(cNormal)
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

ISDIGIT() Ziffer>rmittelt, ob eine Zeichenkette mit einer Ziffer beginnt

---

Syntax

ISDIGIT(<cZeichenkette>) -> lZahl

Parameter

<cZeichenkette> ist die zu untersuchende Zeichenkette.

R.ckgabe

ISDIGIT() liefert true (.T.), wenn das erste Zeichen von <cZeichenkette> eine Ziffer zwischen Null und Neun ist, im anderen Fall wird false (.F.) zur.ckgegeben.

Anwendung

ISDIGIT() ist eine Zeichenfunktion, mit der festgestellt werden kann, ob eine Zeichenkette mit einer Ziffer zwischen Null und Neun beginnt. Ist das erste Zeichen ein anderes Zeichen, liefert ISDIGIT() den Wert false (.F.).

ISDIGIT() ist dann sehr hilfreich, wenn Sie, bevor Sie eine Zeichenfolge mit VAL() in einen numerischen Wert umwandeln, feststellen wollen, ob der angegebene Wert eine numerische Angabe enth.lt.

Beispiele

Das Beispiel zeigt die verschiedenen Ergebnisse der Funktion ISDIGIT():

```
? ISDIGIT("AbcdE")  // Resultat: .F.
? ISDIGIT("1abcd")  // Resultat: .T.
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

ISLOWER() Kleinbuchstabe>ittelt, ob eine Zeichenkette mit einem Kleinbuchstaben beginnt

---

Syntax

ISLOWER(<cZeichenkette>) -> lKlein

Parameter

<cZeichenkette> ist die zu untersuchende Zeichenkette.

R.ckgabe

ISLOWER() gibt true (.T.) zur.ck, wenn das erste Zeichen von <cZeichenkette> ein Kleinbuchstabe ist, ansonsten false (.F.).

Anwendung

ISLOWER() ist eine Zeichenfunktion, welche feststellt, ob eine angegebene Zeichenkette mit einem Kleinbuchstaben (a-z, .-.) beginnt. Die inverse Funktion ist ISUPPER(), welche einen Gro.buchstaben identifiziert.

Beide Funktionen ISLOWER() und ISUPPER() richten sich nach den Funktionen LOWER() und UPPER(), welche Zeichen einer Zeichenkette in Kleinbuchstaben bzw. in Gro.buchstaben umwandeln.

Beispiele

Das Beispiel zeigt die verschiedenen Ergebnisse von ISLOWER():

```
? ISLOWER("aBcDe")  // Resultat: .T.
? ISLOWER("AbCdE")  // Resultat: .F.
? ISLOWER("1abcd")  // Resultat: .F.
? ISLOWER("abcd")   // Resultat: .T.
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

ISPRINTER() Stellt fest, ob LPT1 drucktbereit ist

Syntax

ISPRINTER() -> lBereit

R.ckgabe

ISPRINTER() liefert true (.T.), wenn LPT1 druckbereit ist, ansonsten wird false (.F.) zur.ckgegeben.

Anwendung

ISPRINTER() ist eine Druckerfunktion und wird dazu benutzt, festzustellen, ob der Parallelanschlu. LPT1 online und bereit ist, Zeichen entgegenzunehmen. ISPRINTER() ist eine hardware-abh.ngige Funktion und arbeitet daher nur in BIOS-kompatiblen Systemen.

Bevor eine Druckausgabe an den Drucker gesendet wird, l..t sich .ber ISPRINTER() die Druckbereitschaft feststellen, um Fehlermeldungen zu vermeiden. Wenn ein Fehler w.hrend einer Druckoperation auftritt, wird ein Laufzeitfehler erzeugt.

Beispiele

In diesem Beispiel wird die Druckbereitschaft mit 25 Wiederholungsversuchen getestet. Ist der Druckeranschlu. bereit, beginnt die Druckausgabe:

```
LOCAL nZaehler := 0, nVersuche := 25, lBereit
//
DO WHILE nZaehler++ <= nVersuche .AND. (lBereit := ISPRINTER())
ENDDO
//
IF lBereit
  REPORT FORM Verkauf TO PRINTER
ELSE
  "Drucker nicht bereit..."
  BREAK
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB, Quelldatei ist EXAMPLEA.ASM

See Also:

<cZeichenkette> ist die zu untersuchende Zeichenkette.

Anwendung

Beispiele

Dateien

Parameter

R.ckgabe

@ABSATZTITEL = Siehe auch

Syntax

```
? ISUPPER("$abcd")  // Resultat: .F.
? ISUPPER("8abcd")  // Resultat: .F.
? ISUPPER("aBcDe")  // Resultat: .F.
? ISUPPER("AbCdE")  // Resultat: .T.
ISUPPER(<cZeichenkette>) -> lGro.
```

ISUPPER()

Gro.buchstabe>mittelt, ob eine Zeichenkette mit einem Gro.buchstaben beginnt

---

Beide Funktionen ISUPPER() und ISLOWER() richten sich nach den Funktionen UPPER() und LOWER(), welche Zeichen einer Zeichenkette in Gro.buchstaben bzw. in Kleinbuchstaben umwandeln.
Bibliothek ist CLIPPER.LIB
Das Beispiel zeigt die verschiedenen Ergebnisse von ISUPPER():
ISALPHA(), ISDIGIT(), ISLOWER(), LOWER(), UPPER()
ISUPPER() gibt true (.T.) zur.ck, wenn der erste Buchstabe der Zeichenkette gro.geschrieben ist, im anderen Fall wird false (.F.) zur.ckgegeben.

L2BIN() Konvertiert einen numerischen Wert in einen 32-Bit Bin.rwert

---

Syntax

L2BIN(<nExp>) -> cBin.rwert

Parameter

<nExp> ist der zu konvertierende numerische Wert. Dezimalstellen werden nicht ber.cksichtigt.

R.ckgabe

L2BIN() liefert eine vier Byte lange Zeichenkette als 32-Bit Bin.rwert.

Anwendung

L2BIN() ist eine Low-level Dateifunktion, welche in Verbindung mit FWRITE() benutzt wird, um bin.re 32-Bit Werte in eine DOS-Datei zu schreiben. Diese Funktion ist .hnlich der Funktion I2BIN(), die einen 16-Bit Bin.rwert liefert.

L2BIN() ist die inverse Funktion zu BIN2L(), die ben.tigt wird, um einen numerischen Wert aus einer Bin.rdatei in eine numerische Clipper-Variable zu lesen.

Beispiele

Das Beispiel legt eine neue Bin.rdatei an und schreibt eine Reihe von numerischen Werten mit L2BIN() als 32-Bit Bin.rwert in diese Datei:

```
#include "FileIO.ch"
//
LOCAL nNummer, nKennung
nKennung := FCREATE("Test.dat", FC_NORMAL)
FOR nNummer := 1 TO 100
  FWRITE(nKennung, L2BIN(nNummer))
NEXT
FCLOSE(nKennung)
```

Dateien

Bibliothek ist EXTEND.LIB, Quelldatei ist EXAMPLEA.ASM

See Also:

LASTKEY() Liefert den letzten Tastencode aus dem Tastaturpuffer

---

Syntax

LASTKEY() -> nInkeyCode

R.ckgabe

LASTKEY() liefert eine Zahl zwischen -39 und 386 zur.ck, die dem INKEY()-Wert der zuletzt gedr.ckten Taste entspricht.

Anwendung

LASTKEY() ist eine Tastaturfunktion, welche den INKEY()-Wert des letzten, bereits aus dem Tastaturpuffer abgeholten Tastencodes liefert. Dieser Tastencode kann .ber ACCEPT, INPUT, READ, WAIT(), ACHOICE, DBEDIT(), INKEY() oder MEMOEDIT aus dem Tastaturpuffer geholt worden sein. Der R.ckgabewert von LASTKEY() bleibt so lange erhalten, bis ein anderer Tastencode aus dem Tastaturpuffer geholt wird.

LASTKEY() besitzt in Anwendungsprogrammen eine ganze Reihe von Anwendungsm.glichkeiten wie:

. Ermitteln der Taste, mit der ein READ verlassen worden ist

. Innerhalb einer User Defined Function (UDF) die Taste ermitteln, mit der ein aktuelles GET-Objekt .ber eine VALID-Bedingung verlassen wurde

. Identifizierung einer nicht g.ltigen Taste, die in einer benutzerorientierten Funktion wie ACHOICE(), DBEDIT() oder MEMOEDIT() als letzte gedr.ckt worden ist

LASTKEY() wird in Verbindung mit UPDATE() ben.tigt, um festzustellen, ob ein GET-Objekt Puffer .ber READ ge.ndert wurde.

LASTKEY() ist mit den Funktionen NEXTKEY() und READKEY() verwandt. NEXTKEY() liest den als n.chstes anstehenden Tastencode, ohne ihn aus dem Tastaturpuffer zu l.schen.

Wenn Sie lediglich wissen wollen, ob der Benutzer ein GET-Feld ge.ndert hat, sollten Sie die Funktion UPDATED() benutzen. NEXTKEY() gibt das n.chste Zeichen des Tastaturpuffers an.

Eine Liste der INKEY()-Codes und Inkey.ch-Konstanten f.r jede Taste ist im Anhang C des Referenz-Handbuches 2 enthalten.

Beispiele

Dieses Beispiel zeigt eine typische Anwendung von LASTKEY(), in der ermittelt wird, mit welcher Taste ein READ verlassen worden ist. Der Datenbanksatz wird in der Datenbank nur dann ge.ndert, wenn der Benutzer nicht die [Esc] -Taste gedr.ckt hat und .nderungen in den GET-Feldern vorgenommen worden sind.

```
#Include "Inkey.ch"
//
USE Kunden NEW
cSaldo := Kunden->Saldo
@ 10, 10 SAY "aktueller Saldo: " GET cSaldo
READ
IF (LASTKEY() <> K_ESC) .AND. UPDATED()
                                                                    EPLACE Kunden->Saldo WIT
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist Inkey.ch

See Also:

AADD() F.gt ein neues Element an das Ende eines Arrays an

Syntax

AADD(<aArray>, <expWert>) -> expWert

Parameter

<aArray> ist das Array, an das ein neues Element angef.gt werden soll.

<expWert> gibt den Wert an, mit dem das neue Element zu f.llen ist.

R.ckgabe

AADD() gibt den als Parameter .bergebenen Wert <expWert> zur.ck. Ist <expWert> nicht angegeben, so gibt AADD() immer NIL zur.ck.

Anwendung

AADD() ist eine Arrayfunktion, die das .bergebene Array um ein Element vergr..ert. Dem neuen Element wird der Wert von <expWert> zugewiesen. AADD() wird benutzt, um ein Array dynamisch wachsen zu lassen. Dies kann sehr hilfreich bei der Erstellung von dynamischen Listen sein. Ein gutes Beispiel ist das GETList-Array, welches im Get-System die Liste der aktuellen GET-Objekte enth.lt. Nach einem READ oder einer CLEAR GETS-Anweisung wird GETList geleert. Bei jeder @...GET-Anweisung benutzt das GET-System die Funktion AADD(), um ein neues Element an das Ende des GETList-Arrays anzuh.ngen und ein neues GET-Objekt zuzuweisen.

AADD() ist vergleichbar mit der Funktion ASIZE(), jedoch f.gt AADD() nur jeweils ein Element an, w.hrend ASIZE() ein Array beliebig verg..ern und verkleinern kann. AADD() besitzt den Vorteil, da. dem neuen Element direkt ein Wert zugeordnet werden kann.

AADD() ist ebenfalls vergleichbar mit der Funktion AINS(), welche ein neues Element einf.gt. Die Wirkung ist jedoch unterschiedlich: AINS() bewegt Arrayelemente innerhalb eines Arrays, l..t aber die L.nge des Arrays unbeeinflu.t.

Hinweis

. Ist <expWert> ein weiteres Array, so erh.lt das neue Arrayelement eine Referenz auf das in <expWert> angegebene SubArray.

. Bei fehlendem <expWert> erfolgt keine Fehlermeldung; das Array wird nicht erweitert.

Beispiele

Das folgende Beispiel demonstriert den Effekt mehrerer AADD()-Aufrufe mit einem Array:

```
aArray := {}                  // Resultat: ein leeres Array
AADD(aArray, 5)               // Resultat: ist { 5 }
AADD(aArray, 10)              // Resultat: ist { 5,10 }
AADD(aArray, { 12, 10 } )     // Resultat: ist { 5,10, {12,10} }
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

LASTREC() Ermittelt die Anzahl der Datens.tze in der aktuellen Datenbank

---

Syntax

LASTREC() | RECCOUNT() -> nDatens.tze

R.ckgabe

LASTREC() liefert die Anzahl der physischen Datens.tze der aktuellen Datenbankdatei als ganzzahligen, numerischen Wert. Befehle zum Filtern der Datens.tze, wie SET FILTER oder SET DELETE, haben keinen Einflu. auf das Ergebnis. LASTREC() liefert Null, wenn keine Datenbank im aktuellen Arbeitsbereich ge.ffnet ist.

Anwendung

LASTREC() ist eine Datenbankfunktion, die feststellt, wieviele physische Datens.tze sich in der aktuellen Datenbank befinden. LASTKEY() ist identisch zur Funktion RECCOUNT(), welche aus Kompatibilit.tsgr.nden ebenfalls unterst.tzt wird.

Als Voreinstellung arbeitet LASTREC() im aktuellen Arbeitsbereich. Um diese Funktion auch auf nicht selektierte Arbeitsbereiche anzuwenden, kann ein Aliasbezeichner vorangestellt werden.

Beispiele

. Das folgende Beispiel zeigt die Beziehung zwischen LASTREC(), RECCOUNT() und COUNT:

```
USE verkauf NEW
? LASTREC(), RECCOUNT() // Resultat: 84  84
//
SET FILTER TO Verkaeufer = "1001"
COUNT TO nAnzahl
? nAnzahl, LASTREC()    // Resultat: 14  84
```

. Das n.chste Beispiel zeigt, wie auf einen nichtselektierten Arbeitsbereich zugegriffen wird:

```
USE Verkauf NEW
USE Kunden NEW
? LASTREC(), Verkauf->(LASTREC())
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

LEFT() Ermittelt den linken Teil einer Zeichenkette

Syntax

LEFT(<cZeichenkette>, <nAnzahl>) -> cLinkerTeil

Parameter

<cZeichenkette> ist die Zeichenkette, aus der die ersten Zeichen ermittelt werden sollen. Die maximale Gr..e der Zeichenkette betr.gt 65.520 Zeichen.

<nAnzahl> ist die Anzahl der Zeichen, die, von links beginnend, gelesen werden sollen.

R.ckgabe

LEFT() liefert, von links beginnend, <nAnzahl> Zeichen aus <cZeichenkette> als Zeichenkette. Ist <nAnzahl> negativ oder Null, wird eine leere Zeichenkette zur.ckgegeben.
Ist <nAnzahl> gr..er als Zeichen in <cZeichenkette> sind, liefert LEFT() die komplette Zeichenkette.

Anwendung

LEFT() ist eine Zeichenfunktion, die einen Teil einer .bergebenen Zeichenkette liefert. Sie ist gleich dem Ausdruck:
SUBSTR(<cZeichenkette>, 1, <nAnzahl>). LEFT() ist .hnlich wie RIGHT(), dort wird jedoch der rechte Teil einer Zeichenkette extrahiert.

LEFT(), RIGHT() und SUBSTR() werden oft in Kombination mit den beiden Funktionen AT() und RAT() gebraucht, um die erste oder letzte Position zu bestimmen, bevor ein Teil extrahiert wird.

Beispiele

. Dieses Beispiel ermittelt die ersten drei Zeichen einer Zeichenkette:

```
? LEFT("ABCDEF")  // Resultat: ABC
```

. Das n.chste Beispiel ermittelt den ersten Teil einer Zeichenkette bis zum Vorkommen eines Kommas:

```
LOCAL cName := "Clipper 5.0, Nantucket"
? LEFT(cName, AT(",", cName) - 1)           // Resultat: Clipper 5.0
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

LEN() Ermittelt die L.nge einer Zeichenkette oder die Anzahl von Elementen in einem Array

---

Syntax

LEN(<cZeichenkette> |<aArray>) -> nAnzahl

Parameter

<cZeichenkette> ist die Zeichenkette, deren L.nge ermittelt werden soll.

<aArray> ist das Array, dessen Anzahl von Elementen bestimmt werden soll.

R.ckgabe

LEN() liefert die L.nge einer Zeichenkette oder die Anzahl von Elementen in einem Array als ganzzahligen, numerischen Wert. Ist der .bergebene Parameter eine leere Zeichenkette ("") oder ein leeres Array ({}), wird Null zur.ckgegeben.

Anwendung

LEN() ist eine Zeichen- und Arrayfunktion, die die L.nge einer Zeichenkette oder die Anzahl der Elemente eines Arrays feststellt. In einer Zeichenkette wird jedes Zeichen, einschlie.lich CHR(0), gez.hlt. Eine leere Zeichenkette erzeugt den Wert 0.

In einem Array wird jedes Element gez.hlt. Ist das Array ein mehrdimensionales Array, wird jedes Subarray als ein Element gez.hlt. Dies entspricht der Anzahl der Elemente in der ersten Dimension. Um die Anzahl der Elemente einer anderen Dimension zu bestimmen, ist das Subarray anzugeben (siehe Beispiele). Beachten Sie dabei, da. verschachtelte Arrays in Clipper nicht die gleiche Form haben m.ssen.

Beispiele

. Das Beispiel zeigt LEN() mit verschiedenen Parametern:

```
? LEN("eine Zeichenkette")  // Resultat: 17
? LEN("")                   // Resultat: 0
? LEN(CHR(0))               // Resultat: 1
//
LOCAL aTest[10]
? LEN(arraytest)            // Resultat: 10
```

. Das n.chste Beispiel erzeugt ein mehrdimensionales Array und ermittelt die Anzahl der Elemente des ersten Subarrays:

```
LOCAL aArray := {{ 1, 2},{ 1, 2},{ 1, 2}}
? LEN(aArray)               // Resultat: 3
? LEN(aArray[1])            // Resultat: 2
```

. Das Beispiel zeigt, wie LEN() eingesetzt werden kann, um ein mehrdimensionales Array zu bearbeiten:

```
LOCAL aArray := {{ 1, 2},{ 1, 2},{ 1, 2}}
LOCAL nZeile, nSpalte, nZeilenStand, nSpaltenStand
//
nZeilenStand := LEN(aArray)
FOR nZeile := 1 TO nZeilenStand
  nSpaltenStand := LEN(aArray[nZeile])
  FOR nSpalte := 1 TO nSpaltenStand
    ? nZeile, nSpalte, aArray[nZeile][nSpalte]
  NEXT
NEXT
```

. Dieses Beispiel zeigt eine Funktion, die ein Array mit numerischen Werten liefert, welches den Dimensionen eines, der Funktion .bergebenen, verschachtelten oder mehrdimensionalen Arrays entspricht. Die Funktion geht davon aus, da. die Dimensionen gleich sind:

```
FUNCTION Dimension( aArray )
```

```
  LOCAL aDim := {}
  DO WHILE ( VALTYPE( aArray ))
    AADD( aDim, LEN( aArray ))
    aArray := aArray[1]
  ENDDO
RETURN (aDim)
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

LOG() Ermittelt den nat.rlichen Logarithmus eines numerischen Wertes

---

Syntax

LOG(<nExp>) -> nNaturalLog

Parameter

<nExp> ist ein numerischer Wert gr..er als Null, zu dem der nat.rliche Logarithmus ermittelt werden soll.

R.ckgabe

LOG() liefert den nat.rlichen Logarithmus eines numerischen Wertes. Ist <nExp> gleich oder kleiner Null, erzeugt LOG() einen numerischen .berlauf (erkenntlich an einer Reihe von Sternen).

Anwendung

LOG() ist eine numerische Funktion, die den nat.rlichen Logarithmus eines numerischen Wertes berechnet. Damit stellt LOG() die inverse Funktion zu EXP() dar. Die Basis des nat.rlichen Logarithmus ist e = 2.7183. Die Funktion LOG() berechnet dabei x wie folgt:

e^x = y

wobei y das zu .bergebende Argument von LOG ist (LOG(y) = x). Durch mathematische Rundung ist der R.ckgabewert von LOG() und EXP() nicht immer exakt gleich (z.B. EXP(LOG(x)) ist nicht immer exakt x).

Beispiele

. Das Beispiel zeigt die verschiedenen Ergebnisse von LOG():

```
? LOG(10)        // Resultat: 2.30
? LOG(10 * 2)    // Resultat: 3.00
? EXP(LOG(1))    // Resultat: 1.00
? LOG(2.71)      // Resultat: 1.00
```

. Das Beispiel zeigt eine Funktion, welche den Logarithmus zur Basis 10 berechnet:

```
FUNCTION Log10( nWert )
  IF nWert > 0
    RETURN LOG(nWert) / LOG(10)
  ELSE
    RETURN NIL
  ENDIF
```

Dateien:

Bibliothek ist CLIPPER.LIB

See Also:

LOWER() Konvertiert Gro.buchstaben in Kleinbuchstaben

Syntax

LOWER(<cZeichenkette>) -> cKlein

Parameter

<cZeichenkette> ist die Zeichenkette, die in Kleinbuchstaben konvertiert werden soll.

R.ckgabe

LOWER() liefert eine Kopie von <cZeichenkette>, wobei alle Buchstaben des Alphabets in Gro.buchstaben umgewandelt sind. Alle anderen Zeichen bleiben gleich.

Anwendung

LOWER() ist eine Zeichenfunktion, die alle Buchstaben in einer Zeichenkette in Gro.buchstaben umwandelt. Sie ist verwandt mit der Funktion UPPER(), die alle Buchstaben in Kleinbuchstaben konvertiert. LOWER() ist ebenfalls verwandt mit ISLOWER() und ISUPPER(), die feststellen, ob eine Zeichenkette mit einem Gro.buchstaben oder einem Kleinbuchstaben beginnt. Umlaute werden ber.cksichtigt.

LOWER() dient haupts.chlich zur formatierten Ausgabe einer Zeichenkette. Sie kann f.r Vergleichszwecke herangezogen werden, wo Gro.-/Kleinschreibung nicht ber.cksichtigt werden darf (z.B. in Indexschl.sseln).

Beispiele

Das folgende Beispiel demonstriert die verschiedenen Ergebnisse von LOWER():

```
? LOWER("Hannes")                 // Resultat: hannes
? LOWER("Hannes ist 65 Jahre")    // Resultat: hannes ist 65 jahre
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

LTRIM() Entfernt f.hrende Leerzeichen einer Zeichenkette

Syntax

LTRIM(<cZeichenkette>) -> cZeichenkette

Parameter

<cZeichenkette> ist die Zeichenkette, deren f.hrende Leerzeichen entfernt werden sollen.

R.ckgabe

LTRIM() liefert eine Kopie von <cZeichenkette> ohne f.hrende Leerzeichen. Wird eine leere Zeichenkette ("") oder eine nur mit Leerzeichen gef.llte Zeichenkette .bergeben, liefert LTRIM() eine leere Zeichenkette ("").

Anwendung

LTRIM() ist eine Zeichenfunktion, die zur Formatierung von Zeichenketten angewendet wird. Dies kann zum Beispiel bei Ergebniswerten der Funktion STR() notwendig sein.

LTRIM() ist vergleichbar mit RTRIM(), wo nachfolgende Leerzeichen entfernt werden, und ALLTRIM(), wo sowohl f.hrende als auch nachfolgende Leerzeichen entfernt werden. Im Gegensatz zu ALLTRIM(), LTRIM() und RTRIM() f.gen die Funktionen PADC(), PADL() und PADR() Leerzeichen links und/oder rechts in eine Zeichenkette ein.

Beispiele

Das folgende Beispiel zeigt, wie LTRIM() mit verschiedenen anderen Funktionen eingesetzt werden kann:

```
nNummer := 18
? STR(nNummer)               // Resultat: 18
? LEN(STR(nNummer))          // Resultat: 10
? LTRIM(STR(nNummer))        // Resultat: 18
? LEN(LTRIM(STR(nNummer)))   // Resultat: 2
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

LUPDATE() Ermittelt das Datum der letzten .nderung einer Datenbank

---

Syntax

LUPDATE() -> d.nderung

R.ckgabe

LUPDATE() liefert das Datum der letzten .nderung einer ge.ffneten Datenbank im aktuellen Arbeitsbereich. Ist keine Datenbank mit USE ge.ffnet, wird ein leeres Datum zur.ckgegeben.

Anwendung

LUPDATE() ist eine Datenbankfunktion, die das Datum angibt, an dem die Datenbank des aktuellen Arbeitsbereiches zuletzt ge.ndert und geschlossen wurde.

Als Voreinstellung arbeitet LUPDATE() im aktuellen Arbeitsbereich. Diese Funktion kann durch Voranstellen eines Aliasbezeichners auch auf nichtselektierte Arbeitsbereiche angewendet werden.

Beispiele

. Dieses Beispiel veranschaulicht, da. sich das Datum von LUPDATE() nicht .ndert, bis die Datenbank mit CLOSE geschlossen wird:

```
? DATE()       // Resultat: 01.09.90
USE Verkauf
? LUPDATE()    // Resultat: 31.08.90
//
APPEND BLANK
? LUPDATE()    // Resultat: 31.08.90
CLOSE DATABASES
//
USE Verkauf
? LUPDATE()    // Resultat: 01.09.90
```

. Im n.chsten Beispiel wird gezeigt, wie mit einem Aliasbezeichner auf eine Datenbank in einem anderen Arbeitsbereich zugegriffen werden kann:

```
USE Verkauf NEW
USE Kunden NEW

? LUPDATE(), Verkauf->(LUPDATE())
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

MAX() Liefert den gr..eren von zwei numerischen oder Datumswerten

---

Syntax

MAX(<nExp1>, <nExp2>) -> nMaximum
MAX(<dExp1>, <dExp2>) -> dMaximum

Parameter

<nExp1> und <nExp2> sind zwei zu vergleichende, numerische Werte.
<dExp1> und <dExp2> sind zwei zu vergleichende Datumswerte.

R.ckgabe

MAX() liefert den gr..eren Wert der zwei Parameter. Der Datentyp der R.ckgabe ist identisch mit den .bergebenen Parametern.

Anwendung

MAX() ist eine numerische bzw. Datumsfunktion und wird angewendet, um das Maximum zweier Werte zu bestimmem. Die umgekehrte Funktion zu MAX() ist MIN(), die den kleineren zweier Werte liefert.

Beispiele

. Dieses Beispiel zeigt, wie MAX() den gr..eren zweier Werte liefert:

```
? MAX(1, 2)                    // Resultat: 2
? MAX(2, 1)                    // Resultat: 2
```

. Im folgenden Beispiel werden zwei Datumswerte mit MAX() verglichen:

```
? DATE()                       // Resultat: 01.09.90
? MAX(DATE(), DATE() + 30)     // Resultat: 01.10.90
? MAX(DATE(), CTOD(""))        // Resultat: 01.09.90
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

MAXCOL() Ermittelt die maximal benutzbare Bildschirmspalte

Syntax

MAXCOL() -> nSpalte

R.ckgabe

MAXCOL() liefert die gr..tm.gliche Bildschirmspalte, in der Zeichen dargestellt werden k.nnen.

Anwendung

MAXCOL() ist eine Bildschirmfunktion und wird gebraucht, um die maximal darstellbare Spalte des Bildschirms festzustellen. Beachten Sie, da. COL() und ROW() bei Null beginnen.

Beispiele

Das Beispiel benutzt MAXCOL() und MAXROW(), um innerhalb der Bildschirmgrenzen einen Rahmen zu zeichnen, in dem DBEDIT() ausgef.hrt wird:

```
@ 0, 0 CLEAR
@ 0, 0 TO MAXROW(), MAXCOL() DOUBLE
DBEDIT(1, 1, MAXROW()-1, MAXCOL()-1)
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

MAXROW() Ermittelt die maximal benutzbare Bildschirmzeile

Syntax

MAXROW() -> nZeile

R.ckgabe

MAXROW() liefert die gr..tm.gliche Bildschirmzeile, in der Zeichen dargestellt werden k.nnen.

Anwendung

MAXROW() ist eine Bildschirmfunktion und wird gebraucht, um die maximal darstellbare Zeile des Bildschirms festzustellen. Beachten Sie, da. COL() und ROW() bei Null beginnen.

Beispiele

Das Beispiel zeigt eine Funktion, ScreenSize, die MAXCOL() und MAXROW() benutzt, um den Bildschirmbereich als Array zur.ckzugeben:

```
FUNCTION ScreenSize
RETURN { MAXROW(), MAXCOL() }
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

MEMOEDIT() Anzeigen und/oder editieren von Zeichenketten oder Memofeldern

---

Syntax

```
MEMOEDIT([<cVorgabe>],[<nOben>],[<nLinks>],[<nUnten>],[<nRechts>],
         [<lEditModus>],
         [<cAnwenderfunktion>],
         [<nZeilenL.nge>],
         [<nTabWeite>],
         [<nTextpufferZeile>],
         [<nTextpufferSpalte>],
         [<nFensterZeile>],
         [<nFensterSpalte>]) -> cTextPuffer
```

PARAMETER

<cVorgabe> ist eine Zeichenkette oder ein Memofeld, welches als Vorgabewert in den MEMOEDIT()-Textpuffer kopiert wird. Ist er nicht angegeben, bleibt der Textpuffer leer.

<nOben>, <nLinks>, <nUnten> und <nRechts> sind die obere linke und untere rechte Fensterkoordinate eines Bildschirmbereiches, in dem der Texpuffer angezeigt und editiert werden kann. Die Werte <nOben> und <nUnten> liegen im Bereich von Null bis MAXROW(), die Werte <nlinks> und <nRechts> liegen im Bereich von Null bis MAXCOL(). Sind die Parameter nicht angegeben, werden Maximalwerte angenommen (0, 0, MAXROW(), MAXCOL()).

<lEditModus> legt fest, ob der Textpuffer editiert werden darf, oder ob er lediglich angezeigt werden soll. Spezifiziert mit true (.T.), erlaubt dem Anwender, den Textpuffer zu editieren bzw. zu .ndern. Wird er mit false (.F.) angegeben, erm.glicht er lediglich, im Textpuffer zu bl.ttern. Wird der Parameter nicht angegeben, so gilt als Vorgabe false (.F.).

<cAnwenderfunktion> ist der Name einer User Defined Function, die immer dann aufgerufen wird, wenn der Anwender eine nicht von MEMOEDIT() automatisch verarbeitbare Taste gedr.ckt hat, oder sich keine Tastencodes im Tastaturpuffer befinden. <cAnwenderfunktion> wird als Zeichenkette ohne Klammern und ihre Argumente angegeben. Die Angabe eines logischen Wertes false (.F.) f.r diesen Parameter veranla.t, da. der Vorgabewert <cVorgabe> angezeigt und danach MEMOEDIT() sofort verlassen wird. Ist ein solcher Parameter angegeben, .ndert sich automatisch das Verhalten von MEMOEDIT(). Lesen Sie dazu auch die weitere Beschreibung.

<nZeilenL.nge> pr.ft die L.nge einer, im MEMOEDIT()-Fenster angezeigten Textzeile. Ist eine Textzeile l.nger als mit <nZeilenL.nge> angegeben, wird die Textzeile umgebrochen und der .berstehende Teil in der n.chsten Fensterzeile angezeigt. Ist <nZeilenL.nge> l.nger als die Anzahl der im MEMOEDIT()-Fenster darstellbaren Spalten, rollt der Text in horizontaler Richtung, wenn versucht wird, den Bildchirm-Cursor .ber die R.nder hinaus zu bewegen. Durch Weglassen des Parameters gilt als voreingestellte L.nge : <nRechts> - <nLinks>.

<nTabWeite> gibt die Weite der Tabulatoreinstellung an. Wenn der Anwender die Taste [Tab] bet.tigt, werden Tab-Zeichen in Tabulatorpositionen umgesetzt. Wird <nTabWeite> nicht angegeben, so werden vier Leerzeichen f.r ein Tab-Zeichen eingesetzt.

<nTextpufferZeile> und <nTextPufferSpalte> definieren die Anzeigeposition des Bildschirmcursors innerhalb des Textpuffers, wenn MEMOEDIT() aufgerufen wird. <nTextpufferZeile> beginnt mit Eins und <nTextpufferSpalte> mit Null. Dies sind gleichzeitig die Voreinstellungen, wenn die Parameter nicht angegeben werden.

<nFensterZeile> und <nFensterSpalte> definieren die anf.ngliche Position des Bildschirmcursors innerhalb des MEMOEDIT()-Fensters. Zeilen- und Spaltenpositionen beginnen mit Null. Sind die Parameter nicht angegeben, gilt f.r die anf.ngliche Position im MEMOEDIT()-Fenster die Zeile 0 und die aktuelle Spaltenposition des Cursors.

R.ckgabe

MEMOEDIT() liefert den Textpuffer, wenn der Benutzer den Editiervorgang mit [Strg] + [W] abschlie.t, oder eine Kopie von <cVorgabe>, wenn der Benutzer die Funktion mit [Esc] abgebrochen hat.

Anwendung

MEMOEDIT() ist eine Benutzerschnittstellen-Funktion und wird grunds.tzlich als Editier-Funktion f.r Memofelder oder l.ngere Zeichenketten angewendet. Die Bearbeitung der Texte erfolgt innerhalb eines definierten Bildschirmbereiches, dem MEMOEDIT()-Fenster. .hnlich wie bei anderen Benutzerschnittstellen-Funktionen (ACHOICE(), DBEDIT()) unterst.tzt auch MEMOEDIT() eine Reihe von verschiedenen Modi und schlie.t die Einbindung von selbst definierten Funktionen ein. Diese User Defined Function kann eine Umdefinierung von Bedienungstasten und andere, f.r das Programm relevante Aktivit.ten vornehmen.

. Textpuffer: Wird MEMOEDIT() aufgerufen und der Parameter <cVorgabe> angegeben, so wird dieser Wert in einen Textpuffer kopiert. Der Textpuffer enth.lt den vom Benutzer aktuell eingegebenen bzw. ge.nderten Text. Ist <cVorgabe> nicht angegeben, so zeigt sich dem Benutzer ein leerer Textpuffer.
Verl..t der Benutzer MEMOEDIT() durch Bet.tigung von [Strg]+[W], so wird der Inhalt des bearbeiteten Textpuffers zur.ckgegeben. Bricht der Benutzer den Editiervorgang mit [Esc] ab, wird der Textpuffer verworfen und <cVorgabe> zur.ckgegeben. Dieser R.ckgabewert kann einer Variablen zugeordnet, in ein Memofeld plaziert und als Argu- ment einer weiteren Funktion .bergeben werden.

. Editier-Modi: MEMOEDIT() unterst.tzt zwei Modi, welche durch den Parameter <lEditModus> bestimmt werden. Wenn <lEditModus> den Wert true (.T.) enth.lt, verf.llt MEMOEDIT() in den Edit-Modus und erlaubt dem Benutzer, den Inhalt des Textpuffers zu .ndern. Wird dagegen <lEditModus> mit false (.F.) angegeben, schaltet MEMOEDIT in den Browse-Modus und erlaubt dem Benutzer, lediglich innerhalb des Textpuffers zu bl.ttern, ohne ihn .ndern oder neu eingeben zu k.nnen. Um dem Benutzer das Bl.ttern im Text zu erleichtern, werden die Cursortasten [.] und [.] dazu benutzt, den Text innerhalb des MEMOEDIT()-Fensters um eine Zeile nach oben oder unten zu bewegen.

. Eingabe und .nderung des Textes: Innerhalb von MEMOEDIT() kann der Benutzer durch Positionierung des Cursors, sowie Einf.gen und L.schen von Zeichen den Textpuffer editieren. Zur Unterst.tzung bei der Editierung des Textes sind eine Reihe von verschiedenen Bedienungstasten vorgesehen:

Tabelle 5-20: MEMOEDIT() Bedienungs- und Editiertasten

Taste	Inkey.ch	Aktion
PfeilAufw.rts/Strg+E	K_UP	Eine Zeile h.her
PfeilAbw.rts/Strg+X	K_DN	Eine Zeile tiefer
Linkspfeil/Strg+S	K_LEFT	Ein Zeichen links
Rechtspfeil/Strg+D	K_RIGHT	Ein Zeichen rechts
Strg+Linkspfeil/Strg+A	K_CTRLLEFT	Ein Wort links
Strg+Rechtspfeil/Strg+F	K_CTRLRIGHT	Ein Wort rechts
Pos1	K_HOME	Anfang der aktuellen Zeile
Ende	K_END	Ende der aktuellen Zeile
Strg+Pos1	K_CTRLHOME	Fenster oben links
Strg+Ende	K_CTRLEND	Fenster unten rechts
Bild.	K_PGUP	Ein Bild h.her
Bild.	K_PGDN	Ein Bild tiefer
Strg+Bild.	K_CTRLPGUP	Textanfang
Strg+Bild.	K_CTRLPGDN	Textende
Return	K_RETURN	Anfang der n.chsten Zeile
Entf	K_DEL	Zeichen unter Cursor l.schen
R.ckschritt	K_BS	Zeichen links l.schen
Tab	K_TAB	Tabulator einf.gen
Druckbare Zeichen		Zeichen einf.gen

Strg+Y	K_CTRL_Y	Aktuelle Zeile l.schen
Strg+T	K_CTRL_T	Wort rechts l.schen
Strg+B	K_CTRL_B	Absatz neu formatieren
Einfg/Strg+I	K_INS	Einf.gemodus umschalten
Strg+W	K_CTRL_W	Sichern und beenden
Esc	K_ESC	Abbruch von MEMOEDIT()

Gibt der Benutzer einen Text ein, so hat er zwei Texteingabemodi zur Verf.gung: Einf.gen und .berschreiben. Beim Aufruf von MEMOEDIT() ist der voreingestellte Modus Einf.gen aktiv. Dieser kann in MEMOEDIT() durch Bet.tigen der Taste [Einfg] umgeschaltet werden. Durch mehrmaliges Bet.tigen der Taste [Einfg] wird zwischen den beiden Modi gewechselt. Diese k.nnen auch innerhalb der User Defined Function durch READINSERT() oder R.ckgabe des Wertes 22 umgeschaltet werden. Wenn der Einf.ge-Modus aktiv ist, werden eingegebene Zeichen an der Stelle in den Text eingef.gt, an der sich der Cursor befindet, und der folgende Text nach rechts geschoben. Der Einf.gemodus wird im SCOREBOARD angezeigt. Ist der .berschreiben-Modus aktiv, werden die an der Cursorposition befindlichen Zeichen durch die neu eingegebenen .berschrieben und der folgende Text verbleibt an der aktuellen Position.

Erreicht der Cursor bei der Eingabe des Textes den rechten Rand des MEMOEDIT()-Fensters, wird die aktuelle Zeile in die n.chste Zeile umgebrochen und an der Position im Text ein automatisches Zeilenumbruchzeichen (CHR(141)) eingesetzt. Ist der Parameter <nZeileL.nge> angegeben, wird die Zeile umgebrochen, wenn die L.nge von <nZeilenL.nge erreicht ist. Falls <nZeilenL.nge> gr..er als die Fensterbreite von MEMOEDIT() ist, verschiebt sich der Textinhalt beim Erreichen der Fenstergrenze. Um explizit eine neue Zeile oder einen neuen Absatz zu beginnen, mu. vom Benutzer [Return] eingegeben werden.

. Der Edit-Bildschirm: Die Funktion MEMOEDIT() sichert nicht den Inhalt des Bildschirmausschnittes, wenn er durch das MEMOEDIT()-Fenster .berschrieben wird. Es werden auch keine Rahmen und Titel angezeigt. Diese Erweiterungen m.ssen in einer Prozedur oder einer anwenderdefinierten Funktion erstellt werden, wodurch vielseitigere und individuellere Anwendungsm.glichkeiten bestehen. Sehen Sie dazu auch die Beispiele.

. Die Anwendungsfunktion (UDF): <cAnwenderfunktion> ist eine User Defined Function, die Ausnahmetasten bearbeiten oder die Umkonfiguration spezieller Tasten vornehmen kann. Die Funktion wird zu verschiedenen Zeiten aufgerufen, meist dann, wenn eine Taste bet.tigt wurde, die von MEMOEDIT() nicht behandelt wird. Ausnahmetasten sind alle zur Verf.gung stehenden Alt-, Strg- und Funktionstasten. Wurden die Tasten noch nicht von MEMOEDIT() behandelt, k.nnen diese umkonfiguriert werden. Einige dieser Tasten haben bereits eine vordefinierte Funktion. In der Anwenderfunktion (UDF) k.nnen aufgrund des von MEMOEDIT() mitgeteilten Status verschiedene Aktionen durchgef.hrt und ein Wert zur.ckgegeben werden, der das weitere Verhalten von MEMOEDIT() bestimmt.

See Also:

Wenn der Parameter angegeben ist, bestimmt MEMOEDIT() zwei Klassen von Tasten- codes: Nichtkonfigurierbare- und Ausnahmetasten. Wird eine nichtkonfigurierbare Taste gedr.ckt, wird eine entsprechende Aktion von MEMOEDIT() durchgef.hrt. Anders bei Ausnahmetasten, dort wird die Anwenderfunktion aufgerufen. Ist der Tastaturpuffer leer und stehen somit keine Tastencodes mehr zur Bearbeitung an, erfolgt ein Aufruf der Anwenderfunktion.

Beim Aufruf einer Anwenderfunktion werden automatisch drei Parameter an die aufgerufene Funktion .bergeben. Diese kennzeichnen den aktuellen Modus, die aktuelle Textzeile und die aktuelle Textspalte. Der Modus zeigt den aktuellen Status von MEMOEDIT(), welcher abh.ngig ist von der zuletzt bet.tigten Taste oder der letzten Aktion, die vor Aufruf durchgef.hrt wurde. Folgende Modi sind m.glich:

Tabelle 5-21: MEMOEDIT()-Modi

Modus	Memoedit.ch	Beschreibung
0	ME_IDLE	Wartezustand
1	ME_UNKEY	Ung.ltige Taste, Text unver.ndert
2	ME_UNKEYX	Ung.ltige Taste, Text ver.ndert
3	ME_INIT	Initialisierung

Der Modus 3 bedeutet, da. sich MEMOEDIT() im Initialisierungsmodus befindet. Die Funktion MEMOEDIT() ruft eine angegebene Anwenderfunktion (UDF) sofort auf, nachdem sie selber aufgerufen wurde. Aus diesem Grund kann eine Anforderung zur.ckgegeben werden, die in MEMOEDIT() verschiedene Textformat-Einstellungen vornimmt: Zeilenumbruch, Bl.ttern oder Einf.gen. MEMOEDIT() wiederholt den Aufruf der Anwenderfunktion zu Initialisierungszwecken so lange, bis diese eine Null zur.ckgibt. Danach wird der Inhalt des Textpuffers angezeigt und der Benutzer kann editieren, falls die M.glichkeit .ber <lEditModus> eingestellt wurde. Beachten Sie, da., wenn der Wortumbruch eingeschaltet ist und MEMOEDIT() vom Initialisierungsmodus in den Editiermodus wechselt, der Textpuffer mit <nZeilenL.nge> neu formatiert wird. Um die anf.ngliche Formatierung zu unterdr.cken, kann der Wortumbruch-Schalter nach der Initialisierung ausgeschaltet werden. Beachten Sie ebenfalls, da. die Schalter f.r Bl.ttern und Wortumbruch nicht einer Taste zugeordnet sind, sie k.nnen aber innerhalb der Anwenderfunktion zugewiesen werden.

Die Modi 1 und 2 zeigen an, da. MEMOEDIT() einen nicht erkannten oder konfigurierbaren Tastencode aus dem Tastaturpuffer geholt hat. Konfigurierbare Tastencodes werden bearbeitet durch R.ckgabe des Wertes 0 an MEMOEDIT(), welche die vorgesehene Aktion ausf.hrt. Die R.ckgabe eines anderen Wertes f.hrt zu einer anderen Aktion. Ist die Taste eine MEMOEDIT() nicht bekannte Taste, kann Ihr eine Aktion zugewiesen werden, indem der R.ckgabewert sie anfordert oder indem sie in eigenen Anweisungen definiert wird.

Modus 0 zeigt an, da. MEMOEDIT() nun wartet, da keine Tastencodes zur Bearbeitung ausstehen. Immer wenn MEMOEDIT() in den Wartezustand f.llt, wird die Anwenderfunktion aufgerufen. An dieser Stelle k.nnen Sie Zeilen- und Spaltennummern am Bildschirm anzeigen.

Die zwei anderen Parameter, aktuelle Zeile und Spalte, zeigen die aktuelle Cursorposition innerhalb des Textpuffers an, wenn die Anwenderfunktion aufgerufen wird. Der Zeilenparameter beginnt mit der Position Eins und Spalte beginnt mit Position Null.

Wenn der Modus die Werte 1, 2 oder 3 enth.lt, kann ein Wert zur.ckgegeben werden, der MEMOEDIT() mitteilt, welche Aktion als n.chstes durchzuf.hren ist. Die folgende Tabelle zeigt die m.glichen R.ckgabewerte und die entsprechende Aktion:

Tabelle 5-22: MEMOEDIT() Anwenderfunktion R.ckgabewert

Wert	Memoedit.ch	Aktion

0	ME_DEFAULT	Vorgegebene Aktion durchf.hren
1-31		Angeforderte Programmaktion (entsprechend dem Tastenwert)
32	ME_IGNORE	Taste ignorieren
33	ME_DATA	Taste als Datenwert behandeln
34	ME_TOGGLEWRAP	Wortumbruch ein-/ausschalten
35	ME_TOGGLESCROLL	Scrollmodus ein-/ausschalten
100	ME_WORDRIGHT	Wort-Rechts Operation durchf.hren
101	ME_BOTTOMRIGHT	Text-Ende Operation durchf.hren

. Header-Dateien: Die Modus- und R.ckgabewerte k.nnen einfacher gestaltet werden, wenn Sie die Deklarationen aus der Datei Memoedit.ch verwenden, die sich im Verzeichnis \CLIPPER5\INCLUDE befinden. Im gleichen Verzeichnis befindet sich die Datei Inkey.ch, welche Definitionen f.r die Tastencodes beinhaltet.

Hinweise

. Tastenkonfiguration: Ist der Parameter <cAnwenderfunktion> angegeben, sind die in der folgenden Tabelle aufgef.hrten Tasten konfigurierbar. Ist die Taste konfigurierbar, f.hrt der R.ckgabewert 0 zur vorgesehenen Aktion. Ein anderer R.ckgabewert f.hrt zu der Aktion, die durch die so umdefinierte Taste angegeben wird.

. Wortumbruch: Wortumbruch ist ein Formatierungsmodus, der .ber den R.ckgabewert 34 von der Anwenderfunktion ein- und ausgeschaltet werden kann. Ist der Wortumbruch eingeschaltet (Vorgabe), f.gt MEMOEDIT() ein automatisches Zeilenumbruchzeichen (Soft Carriage-Return/CHR(141)) in den Text ein, wenn ein geschlossenes Wort den Fensterrand bzw. die Zeilenl.nge .berschreitet. Ist der Wortumbruch ausgeschaltet, rollt MEMOEDIT() den Textpuffer solange, bis das Zeilenende erreicht ist. An dieser Stelle mu. der Benutzer die [Return]-Taste bet.tigen, um den Cursor in die n.chste Zeile zu positionieren.

Tabelle 5-23: MEMOEDIT() Konfigurierbare Tasten

Taste	Inkey.ch	Vorgesehene Funktion
Strg+Y	K_CTRL_Y	Aktuelle Zeile l.schen
Strg+T	K_CTRL_T	Wort rechts l.schen
Strg+B	K_CTRL_B	Absatz neu formatieren
Strg+V/Einfg	K_INS	Einf.gemodus ein/aus
Strg+W	K_CTRL_W	Eingabe sichern, MEMOEDIT() verlassen
Esc	K_ESC	MEMOEDIT() abbrechen

. Absatzformatierung: Durch den Tastendruck von [Strg] + [B] oder R.ckgabe von 2 kann der Benutzer den Textpuffer bis zum n.chsten Auftreten einer Zeilenschaltung (hard CR/LF) oder Textende neu formatieren. Dies geschieht ohne R.cksicht darauf, ob der Wortumbruchschalter ein- oder ausgeschaltet ist.

. Automatische Zeilenschaltung: Eine automatische Zeilenschaltung wird zur Darstellung in Ausgabebefehlen wie ? und REPORT FORM oder anderen Textprozessoren ber.cksichtigt. Um ein solches Zeichen (CHR(141)) auszutauschen, stehen Funktionen wie HARDCR(), MEMOTRAN() oder STRTRAN() zur Verf.gung.

. Editieren von Textdateien: MEMOEDIT() kann auch auf separate Textdateien angewendet werden, indem die Textdatei zuvor mit MEMOREAD() in eine Clipper-Variable eingelesen wird. Nach Bearbeitung mit MEMOEDIT() kann diese Variable wieder mit MEMOWRIT() in die Textdatei zur.ckgeschrieben werden.

Beispiele

. Das folgende Beispiel erlaubt dem Benutzer, in einem Memofeld zu bl.ttern, ohne da. er dieses .ndern kann:

```
USE Kunden NEW
SET CURSOR OFF
```

```
MEMOEDIT(Kunden->Notizen, 5, 10, 20, 69, .F.)
SET CURSOR ON
```

. Das n.chste Beispiel zeigt, wie dem Benutzer das Editieren eines Memofeldes erlaubt wird:

```
USE Kunden NEW
REPLACE Kunden->Notizen WITH MEMOEDIT(Kunden->Notizen, 5, 10, 20, 69)
```

. Dieses Beispiel zeigt das Erstellen einer Zeichenkette mittels MEMOEDIT():

```
LOCAL cNotiz
cNotiz := MEMOEDIT()
```

. Dieses Beispiel zeigt eine User Defined Function, die eine Zeichenkette in einem mit Titel versehenen Bildschirmfenster bearbeitet:

```
FUNCTION EditMemo(cString, cTitel, nOben, nLinks, nUnten, nRechts)
LOCAL cScreen := SAVESCREEN(nOben, nLinks, nUnten, nRechts)
  @ nOben -1, nLinks -2 CLEAR TO nUnten + 1, nRechts + 2
  @ nOben -1, nLinks -2 TO nUnten + 1, nRechts + 2
  @ nOben, nLinks SAY "[" + cTitel + "]"
  cString := MEMOEDIT(cString, nOben, nLinks, nUnten, nRechts)
  RESTSCREEN(nOben, nLinks, nUnten, nRechts, cScreen)
RETURN (cString)
```

. Das folgende Beispiel liest den Inhalt einer Textdatei in eine Variable. Diese wird editiert und in die Datei zur.ckgeschrieben:

```
LOCAL cString := MEMOREAD("Text.txt")
cString := MEMOEDIT(cString)
IF .NOT.MEMOWRITE("Text.txt", cString)
  ? "Fehler beim Schreiben der Datei !"
  BREAK
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Dateien sind Memoedit.ch und Inkey.ch

MEMOLINE() Extrahiert eine Textzeile aus einer Zeichenkette oder einem Memofeld

---

Syntax

MEMOLINE(<cZeichenkette>,
[<nZeilenL.nge>],
[<nZeilenNummer>],
[<nTabWeite>],
[<lUmbruch>] -> cZeile

Parameter

<cZeichenkette> ist ein Memofeld oder eine Zeichenkette, aus der eine Textzeile extrahiert werden soll.

<nZeilenL.nge> ist die Anzahl der Zeichen pro Zeile und kann im Bereich von 4 bis 254 angegeben werden. Der Vorgabewert betr.gt 79.

<nZeilenNummer> ist die Nummer der zu extrahierenden Zeile. Der Standardwert betr.gt 1.

<nTabWeite> gibt die Tabulatorweite an. Wird der Wert nicht angegeben, betr.gt der Vorgabewert 4. Ist <nTabWeite> gr..er oder gleich der <nZeilenL.nge>, wird <nTabWeite> automatisch in den Wert <nZeilenL.nge> -1 umgewandelt.

<lUmbruch> schaltet den automatischen Wortumbruch ein oder aus. Die Angabe true (.T.) schaltet den automatischen Wortumbruch ein, false (.F.) schaltet ihn ab. Der voreingestellte Wert ist true (.T.).

R.ckgabe

MEMOLINE() gibt die durch <nZeilenNummer angegebene Textzeile aus <cZeichenkette als Zeichenkette zur.ck. Enth.lt <cZeichenkette> weniger Zeichen als in <nZeilenl.nge> angegeben sind, wird mit Leerzeichen aufgef.llt. Ist <nZeilennummer> gr..er als <cZeichenkette> Zeilen enth.lt, gibt MEMOLINE() eine leere Zeichenkette ("") zur.ck.

Ist <lUmbruch> true (.T.) und w.rde die angegebene Zeilenl.nge ein Wort in der Mitte trennen, wird dieses Wort nicht mit in die zur.ckzugebende Zeichenkette aufgenommen.
Es erscheint dann am Anfang der folgenden Zeile, die mit MEMOLINE() extrahiert wird. Ist <lUmbruch> false (.F.), wird nur die Anzahl von Zeichen zur.ckgegeben, die .ber <nZeilenL.nge> angegeben sind. Die n.chste .ber MEMOLINE() angeforderte Zeile beginnt dann mit dem Zeichen, das der n.chsten Zeilenschaltung (CR/LF) folgt, die dazwischenliegenden Zeichen bleiben unber.cksichtigt.

Anwendung

MEMOLINE() ist eine Memofunktion und wird in Verbindung mit MLCOUNT() benutzt, um Zeilen aus Zeichenketten oder Memofeldern auf der Basis der angegebenen Zeilenl.nge herauszuziehen. Sie ist in Clipper die grunds.tzliche Funktion, um Inhalte aus Memofeldern in Zeichenketten darzustellen.

Um mit dieser Methode zu arbeiten, sollte zun.chst die Anzahl der Zeilen in dem Memofeld oder in der Zeichenkette mit MLCOUNT() ermittelt werden, die dann als obere Grenze einer FOR...NEXT-Schleife angegeben wird. MLCOUNT() ist dabei mit den gleichen Angaben .ber Zeilenl.nge und Tabulatorweite zu versorgen, wie sie auch bei MEMOLINE(), innerhalb der FOR..NEXT-Schleife zum Extrahieren der jeweiligen Zeilen verwendet werden.

Beispiele

Dieses Beispiel zeigt eine grunds.tzliche Methode, um Memofelder und lange Zeichenketten mit Hilfe der Funktionen MEMOLINE() und MLCOUNT() darzustellen:

```
LOCAL nZeilenl.nge := 40, nTabweite := 3, lUmbruch := .T.
LOCAL nZeilen, nAktuell
USE Kunden INDEX Name NEW
//
nZeilen := MLCOUNT(Kunden->Notizen, nZeilenl.nge, nTabweite)
//
SET PRINTER ON
FOR nAktuell := 1 TO nZeilen
  ? MEMOLINE(Kunden->Notizen, nZeilenl.nge, nTabweite, lUmbruch)
NEXT
SET PRINTER OFF
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

ABS() Berechnet den Absolutwert eines numerischen Ausdrucks

Syntax

ABS(<nWert>) -> nAbsolut

Parameter

<nWert> ist der numerische Wert, dessen Absolutwert berechnet werden soll.

R.ckgabe

ABS() gibt einen positiven Wert oder 0 zur.ck, welcher dem Absolutwert von <nWert> entspricht.

Anwendung

ABS() ist eine numerische Funktion und wird benutzt, um den gr..ten Wert einer Zahl, unabh.ngig vom Vorzeichen, festzustellen. Sie erlaubt zum Beispiel, den Unterschied zweier Zahlen festzustellen, ohne zu wissen, welche der beiden Zahlen gr..er ist.

Die numerische Funktion ABS(x) ist wie folgt definiert:

F.r x >= 0 liefert ABS(x) als Ergebnis x, f.r x < 0 liefert ABS(x) als Ergebnis -x.

Beispiele

Die folgenden Beispiele zeigen die typischen Resultate der Funktion ABS():

```
a := 100
b := 150

? a - b                 // Resultat: -50
? ABS(a - b)            // Resultat: 50
? ABS(b - a)            // Resultat: 50
? ABS(-12)              // Resultat: 12
? ABS(0)                // Resultat: 0
```

Dateien

MEMOREAD() Liest den Inhalt einer Datei in eine Zeichenvariable

---

Syntax

MEMOREAD(<cDatei>) -> cZeichenkette

Parameter

<cDatei> ist der Name einer Datei, aus welcher gelesen wird. Der Dateiname mu. mit der Dateinamenserweiterung angegeben werden und kann vorangestellt optional einen Laufwerksbezeichner und einen Pfad enthalten.

R.ckgabe

MEMOREAD() gibt den Inhalt einer Datei als Zeichenkette zur.ck. Die Anzahl von Zeichen, die aus der Datei gelesen werden k.nnen, betr.gt 65.520. Das ist die maximale L.nge einer Zeichenketten-Variablen. Wird die Datei nicht gefunden, gibt MEMOREAD() eine leere Zeichenkette ("") zur.ck.

Anwendung

MEMOREAD() ist eine Memofunktion und dient dazu, Dateien in den Arbeitsspeicher einzulesen, wo dann Ver.nderungen vorgenommen oder die Datei in einem Memofeld abspeichert werden kann. MEMOREAD() wird in Verbindung mit MEMOEDIT() und MEMOWRIT() benutzt, um eine Textdatei zu bearbeiten und wieder zur.ckzuschreiben. MEMOREAD() sucht die Datei <cDatei> zun.chst im aktuellen DOS-Verzeichnis. Wird die Datei dort nicht gefunden, wird sie in der .ber den DOS PATH-Befehl angegebenen Verzeichnisliste gesucht. MEMOREAD() ber.cksichtigt bei der Dateisuche nicht die Clipper-Einstellungen DEFAULT und PATH.

In einer Netzwerkumgebung versucht MEMOREAD() die angegebene Datei im Shared- und Readonly-Modus zu .ffnen. Ist die Datei im exclusiven Zugriff eines anderen Benutzers, liefert MEMOREAD() eine leere Zeichenkette ("").

Beispiele

. Das folgende Beispiel benutzt MEMOREAD() dazu, einem Memofeld "Notiz" und einer Variablen den Inhalt einer Textdatei zuzuweisen:

```
REPLACE Kunden->Notiz WITH MEMOREAD("Temp.txt")
cZeichenkette := MEMOREAD("Temp.txt")
```

. Dieses Beispiel zeigt eine Funktion, welche eine Textdatei editiert:

```
FUNCTION Editor( cDatei )
  LOCAL cString
  IF (cString := MEMOREAD(cDatei)) == ""
    ? "Fehler beim Lesen der Datei :" + cDatei
    RETURN .F.
  ELSE
    MEMOWRIT( cDatei, MEMOEDIT(cString))
    RETURN .T.
  ENDIF
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

MEMORY() Ermittelt die noch verf.gbare Speichergr..e f.r verschiedene Bereiche

---

Syntax

MEMORY(<nExp>) -> nKbyte

Parameter

<nExp> ist ein numerischer Wert, der den Typ des von MEMORY() ermittelten Wertes wie folgt angibt:

Tabelle 5-24: MEMORY() Parameterwerte

Wert	Beschreibung
0	Veranschlagt den totalen freien Raum f.r Zeichenketten
1	Gr..ter zusammenh.ngender Raum f.r Zeichenketten
2	Bereich, der f.r RUN-Aufrufe zur Verf.gung steht

R.ckgabe

MEMORY() liefert den noch belegbaren Speicherplatz als ganzzahligen, numerischen Wert in 1 KByte-Schritten.

Anwendung

MEMORY() ist eine Clipper-Umgebungsfunktion, welche den unterschiedlichen Status des Free pool-Speicherbereiches ermittelt. Free pool ist der dynamische Bereich des Arbeitsspeichers, der zur Speicherung von Zeichenketten und zur Ausf.hrung von RUN-Befehlen benutzt wird.

Beispiele

```
#define MEM_CHAR  0
#define MEM_BLOCK 1
#define MEM_RUN   3
//
IF MEMORY(MEM_RUN) >= 128
  RUN MeinProg
ELSE
  ? "Nicht gen.gend Speicherplatz f.r RUN !"
  BREAK
ENDIF
```

Dateien

MEMOTRAN() Ersetzt Zeilenumbr.che in Zeichenketten oder Memofeldern

---

Syntax

```
MEMOTRAN(<cZeichenkette>,
        [<cHardCr>],
        [<cSoftCr>]) -> cZeichenketteNeu
```

Parameter

<cZeichenkette> ist die Zeichenkette oder das Memofeld, in dem die Zeilenumbruchzeichen getauscht werden sollen.

<cHardCr> ist das Zeichen, das gegen einen harten Zeilenumbruch (Hard-CR/LF) ersetzt werden soll. Der Vorgabewert ist das Semikolon ";".

<cSoftCr> ist das Zeichen, das gegen einen automatischen Zeilenumbruch (Soft-CR/CHR(141)) ersetzt werden soll. Der Vorgabewert ist ein Leerzeichen (" ").

R.ckgabe

MEMOTRAN() liefert eine Kopie von <cZeichenkette>, in der Zeilenschaltungen gegen angegebene Ersatzzeichen getauscht sind.

Anwendung

MEMOTRAN() ist eine Memofunktion, welche in einer langen Zeichenkette oder in einem Memofeld Zeilenschaltungen in eine darstellbare Form tauscht. Die Zeilenschaltungen werden in einer Zeichenkette durch zwei Zeichenkombinationen dargestellt, die durch MEMOEDIT() eingesetzt wurden. Automatische Zeilenumbr.che (CHR(141)) werden eingef.gt, wenn ein Wort nicht mehr in eine Zeile pa.t, harte Zeilenumbr.che (CR/LF) werden eingesetzt, wenn der Benutzer [Return] eingibt.

MEMOTRAN() ist sehr hilfreich, wenn Memofelder in einer REPORT FORM-Anweisung dargestellt werden sollen, die keine automatischen Zeilenumbr.che mit einbezieht. MEMOTRAN() erledigt das durch Austausch der automatischen Umbr.che gegen ein Leerzeichen. Beachten Sie bitte, wenn MEMOTRAN() lediglich in einer REPORT FORM-Anweisung vorkommt, ist diese Funktion in einer EXTERNAL-Anweisung zu deklarieren.

Beispiele

Im folgenden Beispiel werden alle Zeilenumbr.che gegen Leerzeichen ersetzt :

```
REPLACE Notizen WITH MEMOTRAN(Notizen, " ", " ")
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

MEMOWRIT() Schreibt eine Zeichenkette oder ein Memofeld in eine Datei

---

Syntax

MEMOWRIT(<cDatei>, <cZeichenkette>) -> lDurchgef.hrt

Parameter

<cDatei> ist der Name der Datei, in die geschrieben werden soll, einschlie.lich Dateinamenserweiterung und optionaler Angabe von Laufwerksbezeichner und Pfad.

<cZeichenkette> ist die Zeichenkette oder das Memofeld, das in die Datei geschrieben werden soll.

R.ckgabe

MEMOWRIT() liefert true (.T.), wenn die Operation durchgef.hrt werden konnte, ansonsten gibt sie false (.F.) zur.ck.

Anwendung

MEMOWRIT() ist eine Memofunktion, welche Zeichenketten oder Memofelder in eine angegebene Datei schreibt. Wird keine Verzeichnisangabe in <cDatei> vorgenommen, schreibt MEMOWRIT() in das aktuelle DOS-Verzeichnis und nicht in das Verzeichnis, das .ber SET DEFAULT spezifiziert wurde. Eine existierende Datei mit gleichem Dateinamen wird .berschrieben.

MEMOWRIT() wird normalerweise in Verbindung mit MEMOREAD() benutzt, um Dateien in den Speicher zu laden, zu bearbeiten und dann zur.ckzuschreiben.

Beispiele

Dieses Beispiel zeigt, wie MEMOWRIT() in Kombination mit MEMOREAD() benutzt werden kann, um Memofelder mit einem externen Editor zu bearbeiten:

```
LOCAL cEditor := "MyEdit.exe"
USE Kunden NEW
IF MEMOWRIT("ClipTemp.txt", Kunden->Notizen)
  RUN (cEditor + " ClipTemp.txt")
  REPLACE Kunden->Notizen WITH MEMOREAD("ClipTemp.txt")
ELSE
  ? "Fehler beim Schreiben der Datei : ClipTemp.txt"
  BREAK
ENDIF
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

MEMVARBLOCK() Liefert einen SET/GET-Code-Block f.r eine Speichervariable

---

Syntax

MEMVARBLOCK(<cMemvarName>) -> bMemvarBlock

Parameter

<cMemvarName> ist der Name der Speichervariablen, f.r die ein Code-Block erzeugt werden soll.

R.ckgabe

MEMVARBLOCK() liefert einen Code-Block f.r eine Speichervariable. Wird er ausgef.hrt, liefert oder .bergibt er einen Wert von einer oder an eine Speichervariable. Existiert keine Variable mit dem angegebenen Namen, wird NIL zur.ckgegeben.

Anwendung

MEMVARBLOCK() erzeugt einen Code-Block, der zwei Operationen durchf.hren kann. Wird der Code-Block mit einen Parameter ausgef.hrt, weist der Code-Block diesen Wert einer Speichervariablen zu. Wird der Code-Block ohne Parameter ausgef.hrt, liefert er den Wert der Speichervariablen <cMemvar-Name>.

Hinweise

. MEMVARBLOCK() erzeugt lediglich einen Code-Block f.r Speichervariablen, deren Name zur Laufzeit bekannt ist. Daher kann diese Funktion nicht auf LOCAL- und STATIC-Variablen angewendet werden. Die gleiche Restriktion gilt f.r die Erzeugung solcher Code-Bl.cke mit Hilfe des Makro-Operators (&).

Beispiele

. Das folgende Beispiel vergleicht den Einsatz der MEMVARBLOCK()-Funktion mit dem Gebrauch des Makro-Operators. Beachten Sie dabei, da. die Funktion MEMVARBLOCK wesentlich schneller ausgef.hrt wird, da sie nicht den Over-head ben.tigt, der bei Makro-Operationen erforderlich ist:

```
PRIVAT var := "Dies ist ein String"
//
// Ein Set-/Get-Block, erzeugt durch einen Makro-Operator
bSetGet := &( "{|SetVal| IF(SetVal == NIL, var, var := SetVal }" )
//
// Ein Set-/Get-Block, erzeugt durch die MEMVARBLOCK()-Funktion
// der erzeugte Block ist funktionsgleich mit dem obigen Block
bSetGet := MEMVARBLOCK("var")
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

MIN() Ermittelt den kleineren zweier numerischer- oder Datumswerte

---

Syntax

MIN(<nExp1>, <nExp2>) -> nMinExp
MIN(<dExp1>, <dExp2>) -> dMinExp

Parameter

<nExp1> und <nExp2> sind zwei zu vergleichende, numerische Werte.
<dExp1> und <dExp2> sind zwei zu vergleichende Datumswerte.

R.ckgabe

MIN() gibt den kleineren der beiden Ausdr.cke zur.ck. Der R.ckgabewert besitzt den gleichen Datentyp, wie die Argumente.

Anwendung

MIN() ist eine numerische bzw. Datumsfunktion. Sie ist anwendbar in Vergeichsoperationen, um festzustellen, ob ein Wert ein Minimum nicht unterschreitet. Die umgekehrte Funktion zu MIN() ist MAX(), welche den gr..eren zweier .bergebener Werte liefert.

Beispiele

. In diesem Beispiel wird gezeigt, wie MIN() den kleineren Wert zur.ckgibt:

```
? MIN(99, 100)              // Resultat: 99
? MIN(100, 99)              // Resultat: 99
```

. Im n.chsten Beispiel wird die Anwendung von MIN() mit Datumswerten gezeigt:

```
? DATE()                    // Resultat: 01.09.90
? MIN(DATE(), DATE() + 30)  // Resultat: 01.09.90
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

MLCOUNT() Z.hlt die Anzahl der Zeilen in einer Zeichenkette oder in einem Memofeld

---

Syntax

```
MLCOUNT(<cZeichenkette>,
        [<nZeilenL.nge>],
        [<nTabWeite>],
        [<lUmbruch>] -> nZeilen
```

Parameter

<cZeichenkette> ist die Zeichenkette oder das Memofeld, deren Zeilenanzahl bestimmt werden soll.

<nZeilenL.nge> ist die Anzahl der Zeichen pro Zeile und liegt im Bereich zwischen 4 und 254. Ist dieser Wert nicht angegeben, so gilt als Voreinstellung 79.

<nTabWeite> gibt die Tabulatorweite an. Wird der Wert nicht angegeben, betr.gt der Vorgabewert 4. Ist <nTabWeite> gr..er oder gleich <nZeilenL.nge>, wird <nTabWeite> automatisch in den Wert <nZeilenL.nge> -1 umgewandelt.

<lUmbruch> schaltet den automatischen Wortumbruch ein oder aus. Die Angabe true (.T.) schaltet den automatischen Wortumbruch ein, false (.F.) schaltet ihn ab. Der voreingestellte Wert ist true (.T.).

R.ckgabe

MLCOUNT() liefert die Anzahl der Zeilen in <cZeichenkette> auf der Basis der Zeilenl.nge <nZeilenL.nge>, der Tabulatorweite und des Umbruchschalters.

Anwendung

MLCOUNT() ist eine Memofunktion und wird in Verbindung mit MEMOLINE() benutzt, um einzelne Zeilen aus Memofeldern oder langen Zeichenketten darzustellen. Als Basisfunktion liefert sie die Anzahl der in einer Zeichenkette oder in einem Memofeld enthaltenen Zeilen. Dieser Wert kann als Begrenzung in einer FOR...NEXT-Schleife benutzt werden, wenn Zeile f.r Zeile extrahiert werden soll.

Ist <lUmbruch> true (.T.) und w.rde die angegebene Zeilenl.nge ein Wort in der Mitte trennen, wird dieses Wort am Anfang der n.chsten Zeile stehen. Ist <lUmbruch> mit false (.F.) angegeben, z.hlt MLCOUNT() die Zeilen, die mit harten Zeilenumbr.chen abgeschlossen sind.

Beispiele

Dieses Beispiel zeigt den Inhalt der Memofelder Notizen. Ein Memofeld wird dazu in die einzelnen Zeilen zerlegt:

```
LOCAL nZeilenL.nge := 65
USE Kunden NEW
//
DO WHILE .NOT. EOF()
                                nZeilen := MLCOUNT(Kunden->Notizen, nZeilenL.nge)
                                FOR i := 1 TO nZeilen
                                                                                MEMOLINE(
                                NEXT
                                SKIP
                                                                                
ENDDO
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

MLPOS() Ermittelt die Position einer Zeile in einer Zeichenkette oder einem Memofeld

---

Syntax

MLPOS(<cZeichenkette>,<nZeilenL.nge>,<nZeilenNummer>,
[<nTabWeite>],
[<lUmbruch>] -> nPosition

Parameter

<cZeichenkette> ist ein Memofeld oder eine Zeichenkette.

<nZeilenL.nge> ist die Anzahl der Zeichen pro Zeile und kann im Bereich von 4 bis 254 angegeben werden.

<nZeilenNummer> ist die Nummer der zu bestimmenden Zeile.

<nTabWeite> gibt die Tabulatorweite an. Wird der Wert nicht angegeben, betr.gt der Vorgabewert 4. Ist <nTabWeite> gr..er oder gleich <nZeilenL.nge>, wird <nTabWeite> automatisch in den Wert <nZeilenL.nge> -1 umgewandelt.

<lUmbruch> schaltet den automatischen Wortumbruch ein oder aus. Die Angabe true (.T.) schaltet den automatischen Wortumbruch ein, false (.F.) schaltet ihn ab. Der voreingestellte Wert ist true (.T.).

R.ckgabe

MLPOS() liefert die Zeichenposition der .ber <nZeilenNummer> in <cZeichenkette> angegebenen Zeile als numerischen Wert. Ist die Zeilennummer gr..er als die Anzahl der Zeilen in <cZeichenkette>, gibt MLPOS() die L.nge der Zeichenkette an.

Beispiele

Dieses Beispiel demonstriert den Gebrauch von MLPOS(), um die Position einer spezifizierten Zeile zu finden:

```
LOCAL cString := MEMOREAD("Temp.txt"), nZeilenL.nge := 40
nZeile := 5
//
nPosition := MLPOS(cString, nZeilenL.nge, nZeile)
? SUBSTR(cString, nPosition, 12)
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

MOD()* Liefert den dBASE III Plus-Modulus zweier Zahlenwerte

Syntax

MOD (<nDividend>, <nDivisor>) -> nRest

Parameter

<nDividend> ist der Divident f.r die Operation.

<nDivisor> ist der Divisor f.r die Operation.

R.ckgabe

MOD() liefert einen numerischen Restwert der Operation <nDividend> dividiert durch <nDivisor>.

Anwendung

MOD() ist eine numerische Funktion, die die dBASE III Plus-Funktion MOD() emuliert. Sie ist durch die Benutzung des Clipper Modulo-Operators implementiert. Beachten Sie jedoch die Unterschiede zwischen der dBASE III Plus-MOD() Funktion und dem Clipper-Operator, welche in folgender Tabelle aufgef.hrt sind:

Tabelle 5-25: MOD() Modulo-Operator

Divident	Divisor	Modulo-Operator	MOD()	dBASE III Plus - MOD() Funktion
3	0	Error	Error	3
3	-2	1	-1	-1
-3	2	-1	1	1
-3	0	Error	Error	-3
-1	3	-1	2	2
-2	3	-2	1	1
2	-3	2	-1	-1
1	-3	1	-2	-2

MOD ist eine Kompatibilit.tsfunktion und wird daher f.r neue Applikationen nicht empfohlen. Verwenden Sie stattdessen den Modulo-Operator.

Hinweis

. Null Divisor bei dBASE III Plus: In dBASE III Plus liefert ein Null Divisor den Dividenten f.r jeden Wert als Ergebnis. Unter Clipper erzeugt eine Null als Divisor einen Laufzeitfehler.

. Null Divisor in fr.heren Versionen: In Versionen vor Sommer '87 lieferte der Null Divisor ein Null Ergebnis f.r alle Werte des Dividenten. In der Version Sommer '87 und folgenden erzeugt er einen Laufzeitfehler.

Dateien

MONTH() Konvertiert einen Datumswert in den numerischen Wert des Monats

---

Syntax

MONTH(<dDatum>) -> nMonat

Parameter

<dDatum> ist das zu konvertierende Datum.

R.ckgabe

MONTH() liefert den ganzzahligen, numerischen Wert zwischen 0 und 12. Wird ein leeres Datum ("") .bergeben, betr.gt der R.ckgabewert Null.

Anwendung

MONTH() ist eine Konvertierungsfunktion und wird immer dann ben.tigt, wenn der Monat innerhalb einer Berechnung, zum Beispiel in periodischen Auswertungen, als numerischer Wert gebraucht wird. MONTH() ist damit eine der zahlreichen Funktionen, die das Datum in einen numerischen Wert konvertieren. Lesen Sie dazu auch die Beschreibung von DAY(), YEAR() und CMONTH(). CMONTH() ist eine .hnliche Funktion, gibt aber den Monat als Zeichenkette zur.ck.

Beispiele

. Dieses Beispiel beschreibt die R.ckgabewerte von MONTH(), ausgehend vom Systemdatum:

```
? DATE()                // Resultat: 01.09.99
? MONTH(DATE())         // Resultat: 9
? MONTH(DATE()) + 1)    // Resultat: 10
```

. Das n.chste Beispiel beschreibt den R.ckgabewert von MONTH(), wenn ein leeres Datum .bergeben wird:

```
#define LeerDatum  CTOD("")
//
? MONTH(LeerDatum)     // Resultat: 0
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

ACHOICE() Ausf.hrung eines Pop-Up Men.s

Syntax

ACHOICE(<nOben>, <nLinks>, <nUnten>, <nRechts>, <acMen.punkte>,
[<alAusw.hlbar><lAusw.hlbar>],
[<cAnwenderFunktion>],
[<nVorauswahl>],
[<nFensterZeile>]) -> nPosition

Parameter

<nOben>, <nLinks> und <nUnten>, <nRechts> entsprechen den Fensterkoordinaten, innerhalb derer das Pop-Up-Men. erscheinen soll.

<nOben> und <nUnten> sind Zeilenangaben und k.nnen die Werte 0 bis MAXROW() annehmen. <nLinks> und <nRechts> sind Spaltenangaben und k.nnen die Werte 0 bis MAXCOL() annehmen.

<acMen.punkte> ist ein Array, welches die anzuzeigenden Men.punkte in Form von Zeichenketten enth.lt. Die unterschiedlichen Men.punkte werden sp.ter durch eine numerische Position innerhalb des Arrays identifiziert.

<alAusw.hlbar> ist ein paralleles Array mit logischen Werten, welches ausw.hlbare Men.punkte in <acMen.punkte> spezifiziert. Jeweils ein Element in <alAusw.hlbar> spezifiziert ein Element in <acMen.punkte>. Anstelle eines logischen Wertes kann ein Element auch eine Zeichenkette enthalten, welche einen logischen Ausdruck beinhaltet, der w.hrend der Ausf.hrung als Makro ausgewertet wird. Enth.lt ein Element den Wert false (.F.), so bedeutet dies, da. der korrespondierende Men.punkt nicht anw.hlbar ist. Enth.lt ein Element den Wert true (.T.), so kann der entsprechende Men.punkt ausgew.hlt werden. Anstatt eines Arrays mit logischen Werten kann auch ein einzelner logischer Wert als Parameter angeben werden. False (.F.) bedeutet in diesem Falle, da. kein Men.punkt ausw.hlbar ist, true (.T.) bedeutet, alle Men.punkte sind selektierbar. Wird der Parameter nicht angegeben, so sind alle Men.punkte ausw.hlbar.

<cAnwenderFunktion> ist der Name einer User Defined Function (UDF), welche immer dann aufgerufen wird, wenn eine Taste bet.tigt wird. Der Funktionsname wird als Zeichenkette, ohne Klammern und Parameter, angegeben. Beachten Sie, da. das Verhalten von ACHOICE() sich .ndert, wenn der Parameter angegeben wird. N.here Information erhalten Sie weiter unten.

<nVorauswahl> gibt die Position in <acMen.punkte> an, auf dem der hervorgehobene Men.zeiger steht, wenn ACHOICE() aufgerufen wird. Wenn kein oder ein nicht selektierbarer Men.punkt angegeben wird, wird der Men.zeiger auf den ersten Men.punkt gesetzt.

<nFensterZeile> gibt die anf.ngliche, bezogen auf <nOben> relative Fensterzeile an, auf die der Men.zeiger gestellt wird. Die Zeile beginnt mit 0. Als Voreinstellung zeigt sich der Men.zeiger b.ndig mit dem oberen Fensterrand. Dabei kann es vorkommen, da. einige leere Zeilen am unteren Fensterrand erscheinen, auf die der Men.zeiger aber nicht positioniert werden kann. Dieser Funktionsparameter wird benutzt, um die anf.ngliche Position des Men.zeigers im Fenster zu steuern. Dies gilt besonders, wenn mehr Men.punkte vorhanden sind, als das Fenster aufnehmen kann.

Wie bei allen Funktionen k.nnen optionale Parameter durch die Angabe von Kommata weggelassen werden.

R.ckgabe

ACHOICE() liefert als numerisches Ergebnis die ausgew.hlte Position innerhalb von <acMen.punkte>. Wird die Auswahl abgebrochen, liefert ACHOICE() immer 0 zur.ck.

Anwendung

ACHOICE() ist eine anwenderorientierte Funktion, mit der verschiedene Arten von Pop-Up-Men.s erstellt werden k.nnen. Jedes Men. benutzt ein

Array mit Zeichenketten als Men.punkte und ein paralleles Array mit logischen Werten, die bestimmen, ob die jeweils zugeh.rigen Men.punkte ausgew.hlt werden d.rfen. Wird ACHOICE() aufgerufen, werden in einem angegeben Fenster eine Reihe von Men.punkten angezeigt. Dr.ckt der Anwender die Taste [Return], nachdem er eine Auswahl getroffen hat, liefert ACHOICE() die Position des ausgew.hlten Punktes innerhalb des Arrays <acMen.punkte>. Bricht der Anwender die Auswahl durch Bet.tigung der Taste Esc]ab

.berschreitet die Anzahl der Men.punkte in <acMen.punkte> die Anzahl der im angegebenen Fenster darstellbaren Zeilen, so werden die Men.punkte im Fenster gescrollt, wenn der Anwender versucht, den Men.zeiger au.erhalb des oberen oder unteren Rand des Fensters zu bewegen. Beachten Sie dabei, da. dies nur dann der Fall ist, wenn der Men.zeiger den ersten oder letzten Men.punkt noch nicht erreicht hat. Durch Eingabe des Anfangsbuchstabens eines Men.punktes springt der Men.zeiger jeweils auf den Men.punkt, dessen Zeichenketten mit dem entsprechenden Buchstaben beginnen.

. Bedienung des Men.s: ACHOICE() besitzt durch Angeben oder Weglassen des Parameters <cAnwenderFunktion> zwei Arten der Bedienung. Wird der Parameter weggelassen, gelten zur Bedienung die in Tabelle 5-1 aufgef.hrten Tasten.

Tabelle 5-1: ACHOICE() Bedienungstasten ohne UDF

Taste	Inkey.CH	Aktion
PfeilAufw.rts	K_UP	Gehe zum vorhergehenden Men.punkt
PfeilAbw.rts	K_DN	Gehe zum n.chsten Men.punkt
Pos1	K_HOME	Gehe zum ersten Men.punkt
Ende	K_END	Gehe zum letzten Men.punkt
Strg+Pos1	K_CTRLHOME	Gehe zur ersten Fensterzeile
Strg+Ende	K_CTRLEND	Gehe zur letzten Fensterzeile
Bild.	K_PGUP	Gehe zur vorhergehenden Seite
Bild.	K_PGDN	Gehe zur n.chsten Seite
Strg+Bild.	K_CTRLPGUP	Gehe zum ersten Men.punkt
Strg+Bild.	K_CTRLPGDN	Gehe zum letzten Men.punkt
Return	K_RETURN	Auswahl des aktuellen Men.punktes
Esc	K_ESC	Auswahl abbrechen, 0 zur.ckgeben
LinksPfeil	K_LEFT	Auswahl abbrechen, 0 zur.ckgeben
RechtsPfeil	K_RIGHT	Auswahl abbrechen, 0 zur.ckgeben
Buchstabe		Gehe zum n.chsten Men.punkt mit diesem Anfangsbuchstaben

. Farben: Men.punkte werden in der Farbe standard dargestellt, der Men.zeiger mit Hervorhebung und nicht ausw.hlbare Men.punkte in der Farbe unselected.

Die folgende Anweisung

```
SET COLOR TO W+/N, W+/R, , , W/N
```

zeigt ein Men. mit hellem Wei. auf Schwarz; der Men.zeiger ist Hellwei. auf Rot, nicht ausw.hlbare Men.punkte sind normal Wei. auf schwarzem Grund.

. Anwenderfunktion (UDF): Wie bei anderen anwenderorientierten Funktionen unterst.tzt auch ACHOICE() eine User Defined Function. Eine UDF wird dann ben.tigt, wenn ACHOICE() verschachtelt werden soll, um hierarchisch aufgebaute Men.s zu erstellen oder um Bedienungstasten umzudefinieren. Wenn <cAnwenderFunktion> angegeben wird, .ndert sich das Verhalten von ACHOICE(). Haupts.chlich werden weniger Tasten automatisch von ACHOICE() behandelt. Die Kontrolle geht an die Anwenderfunktion .ber, die auf eine bet.tigte Taste reagieren kann. Tasten, die von ACHOICE() behandelt werden, zeigt die folgende Tabelle:

Tabelle 5-2: ACHOICE() Bedienungstasten mit UDF

Taste	Inkey.ch	Aktion
PfeilAufw.rts	K_UP	Gehe zum vorhergehenden Men.punkt
PfeilAbw.rts	K_DN	Gehe zum n.chsten Men.punkt

Strg+Pos1	K_CTRLHOME	Gehe zur ersten Fensterzeile
Strg+Ende	K_CTRLEND	Gehe zur letzten Fensterzeile
Bild.	K_PGUP	Gehe zur vorhergehenden Seite
Bild.	K_PGDN	Gehe zur n.chsten Seite
Strg+Bild.	K_CTRLPGUP	Gehe zum ersten Men.punkt
Strg+Bild.	K_CtrlPGDN	Gehe zur letzten Men.punkt

Wenn ACHOICE() eine Anwenderfunktion aufruft, werden automatisch die drei folgenden Parameter .bergeben:

Der aktuelle ACHOICE()-Modus
Die aktuelle Position des Men.zeigers
Die relative Position der Zeile im Men.fenster.

Der Modus zeigt den aktuellen Status von ACHOICE() an, welcher von der gedr.ckten Taste und von der, von ACHOICE() unternommenen Aktion vor Aufruf der Anwenderfunktion abh.ngt. Der Modus-Parameter hat dabei folgende Bedeutung:

Tabelle 5-3: ACHOICE() Modus

Modus	ACHOICE.CH	Bedeutung
0	AC_IDLE	Ruhezustand
1	AC_HITTOP	Men.zeiger au.erhalb des Men.anfangs
2	AC_HITBOTTOM	Men.zeiger au.erhalb des Men.endes
3	AC_EXCEPT	Nicht behandelter Tastendruck
4	AC_NOITEM	Keine Wahlm.glichkeit

Nachdem die Anwenderfunktion auf die entprechenden Aktionen, welche .ber ACHOICE()-Modus oder LASTKEY() angezeigt wurden, reagiert hat, mu. sie an ACHOICE() einen Wert zur.ckliefern. Dieser Wert teilt ACHOICE() mit, wie die Anwenderfunktion weiter verfahren soll:

Tabelle 5-4: ACHOICE() R.ckgabewert der Anwenderfunktion

Wert	ACHOICE.CH	Aktion
0	AC_ABORT	Auswahl abbrechen, 0 zur.ckgeben
1	AC_SELECT	Auswahl beenden, aktuelle Auswahl zur.ckgeben
2	AC_CONT	Auswahl fortsetzen
3	AC_GOTO	Gehe zum n.chsten Men.punkt, dessen Anfangsbuchstabe der gedr.ckten Taste entspricht

Beispiele

. Das folgende Beispiel benutzt zwei Arrays, um Men.punkte und Auswahlkriterien zu bestimmen. Nachdem ACHOICE() ausgef.hrt und eine Auswahl getroffen worden ist, wird der ausgew.hlte Men.punkt angezeigt:

```
acMen.punkte    := { "Eins", "Zwei", "", "Drei" }
alAusw.hlbar    := { .T., .T., .F., .T. }
nPosition       := ACHOICE( 10, 10, 12, 15, acMen.punkte, alAusw.hlbar)
IF nPosition >u 0
  ? acMen.punkte[ nPosition ]
ENDIF
```

. Das n.chste Beispiel ist .hnlich, jedoch wird eine Anwenderfunktion angegeben und ein Auswahlkriterium ist .ber ein Makro definiert:

```
FUNCTION MyMenu
LOCAL acMen.punkte[4], alAusw.hlbar[4], cAnwenderfunktion := "DoIt"

acMen.punkte[1]   := "Neuer Datensatz"
acMen.punkte[2]   := "Editieren"
acMen.punkte[3]   := "L.schen"
acMen.punkte[4]   := "Speichern"

alAusw.hlbar[1]   := .T.
alAusw.hlbar[2]   := .T.
alAusw.hlbar[3]   := .T.
alAusw.hlbar[4]   := ".NOT.UPDATED()"
```

```
RETURN ACHOICE( 1, 10, 4, 25, acMen.punkte, alAusw.hlbar,;
          cAnwenderfunktion )
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Dateien sind Achoice.ch und Inkey.ch

---

See Also:

NETERR() Pr.ft, ob ein Netzwerkbefehl einen Fehler hervorgerufen hat

---

Syntax

NETERR() -> lFehler

R.ckgabe

NETERR() liefert true (.T.), wenn ein USE oder APPEND BLANK nicht durchgef.hrt werden konnte, im anderen Fall false (.F.). Wenn der vorangegangene Prozess nicht unter einem Netzwerksystem lief, liefert NETERR() immer false (.F.).

Anwendung

NETERR() ist eine Netzwerkfunktion. Sie ist eine globale Zustandsanzeige, welche durch USE, USE...EXCUSIVE und APPEND BLANK in einer Netzwerkumgebung gesetzt wird. Diese Funktion wird ben.tigt, wenn das Fehlverhalten einer dieser Befehle gepr.ft werden soll. Fehler k.nnen durch folgende Situationen auftreten:

Tabelle 5-26: NETERR() Fehlerursachen

Befehl	Ursache
USE	USE...EXCLUSIVE von einem anderen Programm
USE...EXCLUSIVE	USE oder USE...EXCLUSIVE von einem anderen Programm
APPEND BLANK	FLOCK() oder RLOCK() durch ein anderes Programm auf LASTREC() + 1

NETERR() sollte generell nach einem USE oder APPEND angewendet werden, um deren Durchf.hrbarkeit zu pr.fen. Wird false (.F.) zur.ckgegeben, k.nnen die n.chsten Datenbankoperationen durchgef.hrt werden. Lautete der Befehl USE, so k.nnen Indexdateien ge.ffnet werden. Lautete der Befehl APPEND BLANK, so k.nnen dem Datensatz neue Werte mit REPLACE oder @...GET zugewiesen werden. Liefert NETERR() true (.T.), kann auf den Fehler reagiert werden, indem mehrfach die Anweisungen wiederholt werden, oder indem der Anweisungsblock mit BREAK oder RETURN unterbrochen wird.

Beispiele

Dieses Beispiel zeigt die typische Anwendung von NETERR(). Wird USE ohne Fehler ausgef.hrt, k.nnen Indexdateien ge.ffnet werden. Im anderen Fall wird eine Fehlernachricht angezeigt und mit BREAK in die n.chste BEGIN SEQUENCE-Schleife zur.ckgesprungen.

```
USE Kunden NEW
IF .NOT. NETERR()
  SET INDEX TO KundenNr, Name, Ort
ELSE
  ? "Die Kundendatei wird an anderer Stelle bearbeitet."
BREAK
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

NETNAME() Ermittelt den aktuellen Namen einer Workstation

Syntax

NETNAME() -> cWorkstationName

R.ckgabe

NETNAME() liefert die Identifikation einer Arbeitsstation als Zeichenkette mit bis zu 15 Zeichen. Ist als Netzwerksystem kein IBM-PC Netzwerk im Einsatz oder wurde der Name nicht gesetzt, wird eine leere Zeichenkette zur.ckgegeben.

Anwendung

NETNAME() ist eine Netzwerkfunktion. Sie liefert den Namen der Arbeitsstation. Dieser Name wird durch das IBM-PC Netzwerk geliefert. Innerhalb von Novell-Netzwerken k.nnen Sie die Arbeitsplatzbezeichnung in das LOGIN-Script aufnehmen.

Beispiele

. Das Beispiel demonstriert das Ergebnis von NETNAME(), wenn das Programm im Netzwerk auf einer Station mit dem Namen "STATION1" gestartet wurde:

```
? NETNAME()      // Resultat: STATION1
```

. Das n.chste Beispiel zeigt das Ergebnis, wenn das gleiche Programm auf einem Einzelplatzrechner gestartet wird:

```
? NETNAME()      // Resultat: ""
```

Dateien

NEXTKEY() Liest das n.chste, zur Verarbeitung anstehende Zeichen aus dem Tastaturpuffer

---

Syntax

NEXTKEY() -> nInkeyCode

R.ckgabe

NEXTKEY() liefert einen ganzzahligen, numerischen Wert zwischen -39 und 386. Ist der Tastaturpuffer leer, gibt NEXTKEY() null zur.ck. Ist SET TYPEAHEAD auf null gesetzt, gibt NEXTKEY() ebenfalls null zur.ck.

Anwendung

NEXTKEY() ist eine Tastaturfunktion, welche den als n.chstes anstehenden Tastencode aus dem Tastaturpuffer liest, ohne diesen zu entfernen. Der Tastencode entspricht dem Inkey-Code der gedr.ckten Taste, der gleiche Code, der auch von INKEY() oder LASTKEY() zur.ckgegeben wird. NEXTKEY() liefert Werte f.r alle ASCII-Zeichen-, Funktions-, Alt-Funktions-, Strg-Funktions-, Alt-Ziffern-, Strg-Ziffern- und Cursortasten.

NEXTKEY() ist .hnlich der INKEY()-Funktion, unterscheidet sich jedoch in einem wesentlichen Punkt. INKEY() holt das als n.chstes anstehende Zeichen aus dem Tastaturpuffer und aktualisiert den Wert f.r LASTKEY(). Im Gegensatz dazu liest NEXTKEY() lediglich das Zeichen, ohne es aber aus dem Tastaturpuffer zu entfernen, und auch der LASTKEY()-Wert wird nicht aktualisiert.

Eine komplette Liste aller Inkey-Codes ist in der Datei Inkey.ch als Konstantendefinition und im Anhang C dieses Handbuches als Beschreibung aufgef.hrt.

Beispiele

Dieses Beispiel plaziert einen Esc-Tastencode in den Tastaturpuffer und demonstriert den Unterschied zwischen INKEY(), LASTKEY() und NEXTKEY():

```
#include "Inkey.ch"
//
CLEAR TYPEAHEAD
KEYBOARD CHR(K_ESC)
//
? NEXTKEY(),LASTKEY() // Resultat: 27  0
? INKEY(),LASTKEY()   // Resultat: 27  27
? NEXTKEY()       // Resultat: 0
```

Dateien

Bibliothek ist EXTEND.LIB, Header-Datei ist Inkey.ch

See Also:

NOSNOW() Unterdr.ckung von Schnee-Effekten bei Farbbildschirmen

---

Syntax

NOSNOW(<lSchalter>) -> NIL

Parameter

<lSchalter> ist ein logischer Wert, der die Unterdr.ckung von Schnee-Effekten ein- bzw. ausschaltet. Ein logisches TRUE (.T.) schaltet die Unterdr.ckung ein, w.hrend ein logisches FALSE (.F.) die Unterdr.ckung ausschaltet.

Anwendung

Die Funktion NOSNOW() wird bei der Verwendung von einigen CGA-Bildschirmadaptern ben.tigt, um den Schnee-Effekt bei diesen Adaptern zu unterdr.cken. Normalerweise wird diese Funktion im Konfigurationsteil einer Applikation verwendet, um vom Anwender die Unterdr.ckung von Schnee-Effekten einzuschalten, falls der Anwender ein CGA-Bildschirmadapter verwendet.

OS() Liefert den Namen des Betriebssystems

Syntax

OS() -> cOsName

R.ckgabe

OS() liefert den Namen des Betriebssystems als Zeichenkette.

Anwendung

OS() ist eine Environment-Funktion, die den Namen des Betriebssystemes, unter welchem die Applikation l.uft, ermittelt. Der Name enth.lt die Angabe des Systemnamens gefolgt von der Versionsnummer.

Beispiele

Dieses Beispiel benutzt OS(), um festzustellen, auf welchem Betriebssystem die Applikation l.uft:

```
? OS()  // Resultat: DOS 3.3
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

OUTERR() Schreibt eine Reihe von Werten auf die Fehler-Ausgabeeinheit

---

Syntax

OUTERR([<exp liste>]) -> NIL

Parameter

<exp liste> ist eine durch Kommata getrennte Auflistung von Ausdr.cken, die auf der Fehler-Ausgabeeinheit ausgegeben wird. Mit Ausnahme der Datentypen Array und Code-Block k.nnen alle Datentypen angegeben werden.

R.ckgabe

OUTERR() besitzt keinen R.ckgabewert.

Anwendung

OUTERR() ist identisch mit der Funktion OUTSTD(), mit einer Ausnahme: OUTERR() schreibt die Ausgaben in die Standard-Fehlerausgabeeinheit, anstatt auf die Standard-Ausgabeeinheit. Die Ausgabe auf die Standard-Fehlerausgabeeinheit umgeht alle Redirektionen der Clipper Konsolen- und Device-Ausgaben und erst recht allen DOS-Ausgabe-Redirektionen.
Die Funktion OUTERR() wird normalerweise dazu benutzt, Fehlernachrichten auszugeben.

Beispiele

Im diesem Beispiel wird eine Fehlernachricht mit Datum und Uhrzeit auf die Standard-Feherausgabeeinheit ausgegeben:

```
IF NETERR()
  OUTERR("Fehler beim Sperren der Datei! ", DATE(), TIME())
ENDIF
```

See Also:

OUTSTD() Schreibt eine Reihe von Werten auf die Standard-Ausgabeeinheit

---

Syntax

OUTSTD([<exp liste>]) -> NIL

Parameter

<exp liste> ist eine durch Kommata getrennte Auflistung von Ausdr.cken, die auf der Standard-Ausgabeeinheit ausgegeben wird. Mit Ausnahme der Datentypen Array und Code-Block k.nnen alle Datentypen angegeben werden.

R.ckgabe

OUTERR() besitzt keinen R.ckgabewert.

Anwendung

OUTSTD() ist eine einfache Ausgabefunktion. Die Funktion ist .hnlich wie QOUT(), schreibt aber die Werte auf die Standard-Ausgabeeinheit anstatt auf die Clipper-Konsolenausgabe. Programme mit einfachen Ausgaben (keine FULL-SCREEN-Ausgaben) k.nnen diese Funktion anstelle des TERMINAL-Subsystems benutzen. Die Header-Datei SIMPLIO.CH redefiniert die ?- und ??-Ausgabe-befehle mit Hilfe der OUTSTD()-Funktion.

Da OUTSTD() die Ausgaben auf die Standard-Ausgabeeinheit (CON:) ausgibt, reagieren sie auf DOS-Redirektions-Symbole (<, > und |). Dies erlaubt die Umlenkung von Clipper-Ausgaben in Dateien oder auf andere Einheiten. Weiter-gehende Informationen erhalten Sie auch in Ihrer PC-/MS-DOS-Dokumentation.

Beispiele

. Das folgende Beispiel benutzt OUTSTD(), um eine Reihe von Ausdr.cken auf die Standard-Ausgabeeinheit auszugeben:

```
OUTSTD(Name, PADR(RTRIM(Ort),20),",",PLZ)
```

. Das Beispiel zeigt die Redirektion einer Ausgabe von einem Clipper-Programm in eine Datei DATEI.TXT:

```
C:> MeinProg > DATEI.TXT
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist SIMPLIO.CH

See Also:

PAD() F.llt eine Zeichenkette am Anfang und/oder Ende mit Zeichen

---

Syntax

PADL(<exp>, <nL.nge>, [<cF.llZeichen>]) -> cGef.llt
PADC(<exp>, <nL.nge>, [<cF.llZeichen>]) -> cGef.llt
PADR(<exp>, <nL.nge>, [<cF.llZeichen>]) -> cGef.llt

Parameter

<exp ist eine Zeichenkette, die mit einem bestimmten Zeichen auf <nL.nge> aufgef.llt wird.

<nL.nge> ist die L.nge der zur.ckgegebenen Zeichenkette.

<cF.llZeichen> ist das Zeichen, mit dem <exp> zu f.llen ist. Wird der Parameter nicht angegeben, so wird mit Leerzeichen gef.llt.

R.ckgabe

PADL(), PADC() und PADL() liefern das Ergebnis aus <exp>, aufgef.llt mit <cF.llZeichen> als Zeichenkette in der L.nge <nL.nge>.

Anwendung

PADL(), PADC() und PADL() sind zeichenorientierte Funktionen und dienen zum Auff.llen einer Zeichenkett mit einem F.llzeichen, um eine neue, formatierte Zeichenkette in der mit <nL.nge> angegebenen L.nge zu bilden. PADC() zentriert den Ausdruck <exp> und f.llt ihn links und rechts mit F.llzeichen. PADL() f.llt mit Zeichen am linken Rand und PADR() f.llt mit Zeichen am rechten Rand. .berschreitet die L.nge in <exp> den Wert in <nL.nge>, wird die zur.ckgegebene Zeichenkette auf die L.nge <nL.nge> beschnitten und nicht mit Zeichen gef.llt.

PADL(), PADC() und PADL() werden ben.tigt, um in der L.nge variable Werte in festen L.ngen darzustellen. Sie k.nnen in Anwendungsbereichen verwendet werden, wo mit ??-Befehlen Bildschirm- oder Druckerausgaben ben.tigt werden. Eine andere Anwendung besteht darin, feste Bildschirmbereiche mit Texten, die in variablen L.ngen vorkommen, komplett zu .berschreiben.

PADC(), PADL() und PADR() sind die umgekehrten Funktionen zu ALLTRIMM(), LTRIM() und RTRIM, welche f.hrende und nachfolgende Leerzeichen in Zeichenketten entfernen.

Beispiele

Dieses Beispiel benutzt PADR(), um eine formatierte Ausgabe in der Statuszeile zu erzeugen:

```
IF EOF()
  @ 23, 45 SAY PADR( "EOF/" + LTRIM(STR(LASTREC())), 20)
ELSEIF BOF()
  @ 23, 45 SAY PADR( "BOF/" + LTRIM(STR(LASTREC())), 20)
ELSE
  @ 23, 45 SAY PADR( "Rec:" + LTRIM(STR(RECNO())), 20)
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

PCOL() Gibt die aktuelle Position des Druckkopfes zur.ck

Syntax

PCOL() -> nSpalte

R.ckgabe

PCOL() gibt einen ganzzahligen, numerischen Wert zur.ck, der in der Druckposition der aktuellen Spalte des zuletzt gedruckten Zeichens plus eins liegt. Die Spalten beginnen dabei mit 0.

Anwendung

PCOL() ist eine Druckerfunktion und ermittelt die Position des Druckkopfes nach der letzten Druckausgabe. Der Wert von PCOL() wird nur dann aktualisiert, wenn eines der Kommandos SET PRINT ON oder SET DEVICE TO PRINT den Drucker eingeschaltet hat. PCOL() ist der Funktion COL() gleichzustellen, wobei anstatt der Cursorposition auf dem Bildschirm die Position des Druckkopfes ermittelt wird. PCOL() wird auf folgende Weise aktualisiert:

. Programmstart setzt PCOL() auf Null
. EJECT setzt PCOL() auf Null zur.ck
. Eine Druckoperation setzt PCOL() auf die letzte Druckposition + 1
. SETPRC() setzt PCOL() auf den spezifizierten Wert

PCOL() wird zusammen mit PROW() benutzt, um Druckausgaben relativ zur letzten Druckposition in der gleichen Zeile zu bestimmen. Dies erleichtert die Darstellung tabellenartiger Reports. Ein Wert wird dabei durch die Berechnung PCOL() + <SpaltenOffset> in die n.chste Spalte gedruckt. Um die ausgegebenen Werte gleich zu formatieren, k.nnen die Funktionen TRANSFORM() oder PAD() bzw. die PICTURE-Anweisung von @...SAY benutzt werden.

Hinweise

. Drucker Steuerzeichen: Die Ausgabe von Steuerzeichen f.r den Drucker kann dazu f.hren, da. die aktuelle Druckkopfposition nicht mehr mit PCOL() .bereinstimmt.
Die Diskrepanz wird verursacht, da Clipper alle Zeichen z.hlt, die zur Druckausgabe gelangen, Steuerzeichen, die den Druckkopf positionieren, jedoch nicht ber.cksichtigt werden. Um solchen Schwierigkeiten entgegenzutreten, sollten die Werte aus PCOL() und PROW() gesichert werden, bevor Steuerzeichen an den Drucker geschickt werden, um sie anschlie.end .ber SETPRC() wieder zu synchronisieren.

. SET MARGIN: PCOL() kann nicht zuverl.ssig mit der Kombination SET MARGIN und dem Druckbefehl @...SAY zusammenarbeiten. Bevor Zeichen .ber @...SAY gedruckt werden, wird der aktuelle MARGIN-Wert zu der jeweiligen Spaltenposition hinzu addiert. Dieser Effekt f.hrt dazu, da. die Werte von MARGIN und PCOL() addiert werden, bevor mit @...SAY eine Ausgabe in der gleichen Zeile erfolgt.

Beispiele

Dieses Beispiel benutzt PCOL(), um einen einfachen Ausdruck zu erzeugen, der die Namen, Adressen und Telefonnummern der Kunden enth.lt:

```
LOCAL nZeile := 99, nSeite := 1
USE Kunden INDEX Name NEW
SET DEVICE TO PRINTER
//
DO WHILE .NOT.EOF()
  IF nZeile > 55
    PageTop(nSeite)
    nZeile := 1
    nSeite++
  ENDIF
  @ nZeile, 10 SAY Kunden->Name
  @ nZeile, PCOL() + 2 SAY RTRIM(Kunden->Land) + ", ";
    + RTRIM(Kunden->Plz) + Kunden->Stadt
```

```
  @ nZeile, PCOL() + 2 SAY Kunden->Telefon PICTURE "99999/999999"
  nZeile++
  SKIP
ENDDO
SET DEVICE TO SCREEN
CLOSE
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

PCOUNT() Ermittelt die Anzahl der .bergebenen Parameter

Syntax

PCOUNT() -> nAnzahlParameter

R.ckgabe

PCOUNT() gibt die Anzahl der .bergebenen Parameter als ganzahligen, numerischen Wert zur.ck. Wird kein Parameter .bergeben, liefert PCOUNT() Null.

Anwendung

PCOUNT() stellt die Anzahl der Parameter fest, wenn eine Prozedur oder eine User Defined Function aufgerufen wird. Mit dieser Information kann festgestellt werden, ob optionale Parameter am Ende der Parameterliste weggelassen wurden. Parameter, die in der Mitte der Parameterliste weggelassen wurden (Parameter Skipping), werden mitgez.hlt.

Um festzustellen, ob ein Parameter weggelassen wurde, sollte er auf NIL getestet werden. Parameter, die weggelassen werden, sind nicht initialisiert und ergeben den Typ NIL, wenn darauf zugegriffen wird. Eine andere Testmethode liefert die Funktion VALTYPE(). Diese kann angewendet werden, wenn .bergebene Parameter einem korrekten Typ entsprechen sollen.

Beispiele

. Dieses Beispiel ist eine User Defined Function, die versucht, eine Datenbank zu .ffnen und dabei PCOUNT() benutzt, um zu pr.fen, ob der Dateiname angegeben worden ist. Wurde der Name nicht angegeben, fragt die Funktion danach:

```
FUNCTION OeffneDbf(cDatei)
                                  IF PCOUNT() = 0
.ACCEPT "Geben Sie den Datenbanknamen ein: " TO cDatei
                                  ENDIF
                                  USE (cDatei)
RETURN (NETERR())
```

. Das n.chste Beispiel demonstriert, wie .bergebene Parameter .berpr.ft werden k.nnen:

```
MeineFunkt("Parameter1", , ,12)
//
FUNCTION MeineFunkt(par1, par2, par3, par4)
  ? PCOUNT()                      // Resultat: 4
  ? par2 = NIL, par4 = NIL        // Resultat: .T. .F.

RETURN
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

PROCLINE()
  Ermittelt die aktuelle Zeilennummer im Quellprogramm

Syntax
  PROCLINE([<nAktivierung>]) -> nQuellZeile

Parameter
  <nAktivierung> ist ein numerischer Wert, welcher den Programmabschnitt
  angibt, zu dem die Zeilennummer des Quellprogammes ermittelt werden soll.
  Null zeigt auf den aktuellen Programmabschnitt, eins auf den
  vorangegangenen Programmabschnitt usw. Ist der Parameter nicht angegeben,
  betr.gt der Vorgabewert Null.

R.ckgabe
  PROCLINE() liefert die Zeilennummer der zuletzt ausgef.hrten Zeile in
  der aktuellen Prozedur, Funktion oder des Code-Blocks als numerischen,
  ganzzahligen Wert. Wurde die /L-Option beim Compilieren der Programme
  verwendet, liefert PROCLINE() immer den Wert Null.

Anwendung
  PROCLINE() fragt den Clipper Activation Stack, welche die letzte
  Zeilennummer war, die in der aktuellen Prozedur, User Defined Function
  oder in einem Code-Block ausgef.hrt wurde. Der Clipper Activation Stack
  ist eine interne Struktur, die f.r jeden Prozedur-, Funktions- und
  Code-Blockaufruf einen Datensatz (activation record) enth.lt. Eine
  Zeilennummer ist immer relativ zum Anfang einer Quelldatei. Eine Zeile
  schlie.t Kommentar-, Leer-, Pre-Prozessor- oder Compiler-Anweisungszeilen
  mit ein. Sind mehrere Anweisungen in einer Zeile enthalten, so wird nur
  eine Zeile gez.hlt.

  F.r einen aktuellen Programmabschnitt liefert PROCLINE() die Nummer der
  aktuellen Zeile. F.r einen vorangegangenen Programmabschnitt liefert
  PROCLINE() immer die Zeilennummer, von welcher die aktuelle Prozedur,
  Funktion oder der Code-Block aufgerufen wurde.

  Ist der erfragte Programmabschnitt eine Code-Block Berechnung, liefert
  PROCLINE() immer die Zeile, in der der Code-Block definiert wurde.

  PROCLINE() wird zusammen mit PROCNAME() zur Fehlersuche verwendet.

Beispiele
  Das Beispiel zeigt, wie PROCLINE() Zeilennummern f.r den aktuellen und
  f.r den vorangegangenen Programmabschnitt ermittelt:

```
  // Erste Zeile des Quellprogrammes
  MeineFunkt()
  RETURN


  FUNCTION MeineFunkt
    ? PROCLINE()    // Resultat: 6 (aktueller Programmabschnitt)
    ? PROCLINE(1)   // Resultat: 2 (vorheriger Programmabschnitt)
  RETURN NIL
```

Dateien
  Bibliothek ist CLIPPER.LIB

See Also:

PROCNAME()
  Gibt den Namen der gerade ausgef.hrten Prozedur zur.ck

Syntax
  PROCNAME([<nAktivierung>]) -> cProzedurName

Parameter
  <nAktivierung> ist ein numerischer Wert, welcher den Programmabschnitt
  angibt, zu dem die Zeilennummer des Quellprogammes ermittelt werden soll.
  Null zeigt auf den aktuellen Programmabschnitt, Eins auf den
  vorangegangenen Programmabschnitt usw. Ist der Parameter nicht angegeben,
  betr.gt der Vorgabewert Null.

R.ckgabe
  PROCNAME() ermittelt den Namen der momentan ausgef.hrten Prozedur,
  Funktion oder des Code-Blocks als Zeichenkette.

Anwendung
  PROCLINE() fragt den Clipper Activation Stack, welchen Namen die aktuell
  ausgef.hrte Prozedur, User Defined Function oder ein Code-Block besitzt.
  Der Clipper Activation Stack ist eine interne Struktur, die f.r jeden
  Prozedur-, Funktions- und Code-Blockaufruf einen Datensatz (activation
  record) enth.lt.

  F.r einen aktuellen Programmabschnitt liefert PROCNAME() den Namen der
  aktuell ausgef.hrten Prozedur oder User Defined Function. F.r einen
  vorangegangenen Programmabschnitt liefert PROCNAME() immer den Namen der
  Prozedur oder User Defined Function, welche die aktuelle Prozedur oder
  Funktion aufgerufen hat.

  Ist der Programmabschnitt ein Code-Block, liefert PROCNAME() immer den
  Namen der Prozedur oder der User Defined Function, in welcher der
  Code-Block definiert wurde.

Beispiele
  Dieses Beispiel ist eine User Defined Function, die w.hrend der
  Fehlersuche aufgerufen werden kann, um den Activation Stack mit
  Zeilennummern anzuzeigen:

```
FUNCTION ListStack( cNachricht )
  LOCAL nProgSegment := 0
  ? cNachricht
  DO WHILE .NOT. (PROCNAME(nProgSegment) == "")
    ? "Aufgerufen von: " + PROCNAME(nProgSegment),;
      "(" + STR(PROCLINE(nProgSegment), 5) + ")"
    nProgSegment++
  ENDDO
  RETURN NIL
```

Dateien
  Bibliothek ist CLIPPER.LIB

See Also:

PROW() Ermittelt die aktuelle Zeilenposition des Druckkopfes

Syntax

PROW() -> nZeilenNummer

R.ckgabe

PROW() gibt die Nummer der Zeile, in der sich der Druckkopf gerade befindet, als ganzzahligen, numerischen Wert zur.ck. Die Zeilennummerierung beginnt bei Null.

Anwendung

PROW() ist eine Druckerfunktion und ermittelt die Zeilenposition des Druckkopfes nach der letzten Druckoperation. Der Wert von PROW() wird nur dann aktualisiert, wenn eine der Anweisungen SET PRINT ON oder SET DEVICE TO PRINT die Druckausgabe eingeschaltet hat. PROW() ist eine .hnliche Funktion wie ROW(), welche die aktuelle Zeile des Bildschirms ermittelt. Wie der Wert von PCOL() gesetzt wird, geht aus der folgenden Aufstellung hervor.

. Programmstart setzt PROW() auf Null
. EJECT setzt PROW() auf Null zur.ck
. Eine Druckoperation setzt PROW() auf die letzte Druckzeilenposition
. SETPRC() setzt PROW() in die angegebene Zeilenposition

PROW() wird in Kombination mit PCOL() benutzt, um eine Druckausgabe relativ zur letzten Druckposition zu erzeugen. Wird die Druckkopfpositionierung .ber Steuersequenzen vorgenommen (z.B. LF/CHR(10) oder FF/CHR(12)), wird der Wert in PROW() nicht richtig aktualisiert und es kann zu unerwarteten Ergebnissen kommen. Um solche Diskrepanzen zu umgehen, k.nnen mit SETPRC() die korrekten Werte gesetzt werden.

Beispiele

Das Beispiel zeigt, wie PROW() und SETPRC() benutzt werden, um ein einfaches Etikettierprogramm mit @...SAY zu schreiben:

```
USE Kunden INDEX Namen NEW
SET DEVICE TO PRINTER
SETPRC(2, 0)
DO WHILE .NOT.EOF()
  @ PROW(), 5 SAY Name
  @ PROW() + 1, 1 SAY Plz + Stadt
  @ PROW() + 1, 5 SAY "Telefon :"
  @ PROW() , PCOL() + 1 SAY Telefon PICTURE "99999/99999999"
  SETPRC(2, 0)
  SKIP
ENDDO
SET DEVICE TO SCREEN
CLOSE
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

QOUT() Anzeige einer Reihe von Ausdr.cken auf der Konsole

Syntax

QOUT([<exp liste>]) -> NIL
QQOUT([<exp liste>]) -> NIL

Parameter

<exp liste> ist eine durch Komma getrennte Auflistung von Ausdr.cken, die auf der Konsole ausgegeben wird. Mit Ausnahme der Datentypen Array und Code-Block k.nnen alle Datentypen angegeben werden. Wird an die Funktion QOUT() kein Parameter .bergeben, gibt die Funktion lediglich eine CR+LF (Carriage Return + Line Feed) Sequenz aus. QQOUT() ohne Parameter erzeugt keine Ausgabe auf der Konsole.

R.ckgabe

QOUT() und QQOUT() besitzen keinen R.ckgabewert.

Anwendung

QOUT() und QQOUT() sind Konsolenfunktionen. Sie sind die Basisfunktionen, mit denen die Befehle ? und ?? in Clipper realisiert sind. Genau wie bei diesen Befehlen werden diese Funktionen ben.tigt, um Resultate eines oder mehrerer Ausdr.cke auf der Konsole auszugeben.
QOUT() gibt eine CR/LF Sequenz aus, bevor die Resultate aus <exp liste> ausgegeben werden. QQOUT() gibt die Resultate an der .ber ROW() und COL() gesetzten Position aus. Nachdem .ber QOUT() und QQOUT() eine Ausgabe auf der Konsole erfolgt ist, werden die Werte in COL() und ROW() aktualisiert. Werden .ber SET PRINTER ON die Ausgaben auf den Drucker gelenkt, erfolgt stattdessen eine Aktualisierung von PCOL() und PROW(). Mehrere Ausdr.cke in <exp liste> werden durch Leerzeichen zwischen den einzelnen Ausgaben getrennt.

QOUT() und QQOUT() k.nnen innerhalb von Ausdr.cken Ausgaben erzeugen. Diese M.glichkeit ist hilfreich bei sequentiell arbeitenden Funktionen wie AEVAL() und DBEVAL(), sowie in Statements zur Ausgabedefinition bei User Defined Commands.

Beispiele

Das folgende Beispiel demonstriert, wie QOUT() mit AEVAL() dazu benutzt werden k.nnen, Inhalte literaler Arrays auf der Konsole auszugeben:

```
LOCAL aElemente := { 1, 2, 3, 4, 5 }
AEVAL(aElemente, {|Element| QOUT(Element) })
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

RAT() Ermittelt das letzte Auftreten einer gesuchten Zeichenfolge

---

Syntax

RAT(<cSuchfolge>, <cZeichenkette>) -> nPosition

Parameter

<cSuchfolge> ist die gesuchte Zeichenfolge.

<cZeichenkette> ist die zu durchsuchende Zeichenkette.

R.ckgabe

RAT() liefert die Position von <cSuchfolge> in der Zeichenkette <cZeichenkette> als ganzzahligen, numerischen Wert. Ist die Suchfolge nicht in der Zeichenkette enthalten, so liefert sie den Wert Null.

Anwendung

RAT() ist eine zeichenorientierte Funktion, welche die Position des letzten Auftretens einer Suchfolge in einer Zeichenkette als numerischen Wert ermittelt. Die Suche in <cZeichenkette beginnt also rechts. RAT() ist .hnlich der Funktion AT(), mit der das erste Auftreten einer Suchfolge bestimmt wird. RAT() ist auch .hnlich dem $-Operator, welcher .berpr.ft, ob eine Zeichenfolge in einer Zeichenkette enthalten ist.

Die beiden Funktionen RAT() und AT() werden meist in Kombination mit den Funktionen SUBSTR(), LEFT() und RIGHT() benutzt, um bestimmte Zeichenfolgen zu extrahieren.

Beispiele

Dieses Beispiel verwendet RAT() in einer User Defined Function mit dem Namen FilePath(), die den Verzeichnisnamen aus einer Dateispezifikation ermittelt:

```
? FilePath("C:\Dbf\Verkauf.dbf")  // Resultat: C:\Dbf\

FUNCTION FilePath( cDateiName )
  LOCAL nPos, cVerzeichnis := ""
  IF (nPos := RAT("\", cDateiName ) <> 0
    cVerzeichnis := SUBSTR(cDateiName, 1, nPos)
  ENDIF
RETURN cVerzeichnis
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

READEXIT() Ein-/Ausschalten der Tasten Pfeilauf undPfeilab zum Verlassen eines READ

Syntax

READEXIT([<lSchalter>]) -> lSchalterAlt

Parameter

<lSchalter> schaltet die Tasten [.] und [.] zum Verlassen eines READs ein oder aus. Die Angabe von true (.T.) schaltet die Tasten zum Verlassen ein, false (.F.) schaltet sie aus. Wird der Parameter nicht angegeben, erfolgt keine Beeinflussung der aktuellen Einstellung.

R.ckgabe

READEXIT() gibt die aktuelle Einstellung als logischen Wert zur.ck.

Anwendung

READEXIT() ist eine Environment-Funktion, die den aktuellen Status ermittelt, welcher das Verlassen eines ersten oder letzten GET-Objektes innerhalb einer READ-Anweisung mit den Tasten [.] bzw. [.] bestimmt. Wird der Parameter <lSchalter> angegeben, k.nnen die Tastenfunktionen gesperrt oder freigegeben werden. Nach dem Programmstart sind diese Tastenfunktionen gesperrt. Normalerweise kann ein READ nur .ber die Tasten [Bild.], [Bild.] ,[Esc] oder im letzten GET mit [Return] verlassen werden.

Beispiele

In diesem Beispiel wird die Einstellung von READEXIT() gerettet, auf true geschaltet und zum Schlu. in ihren fr.heren Zustand versetzt:

```
cMyVar := SPACE(10)
lLastExit := READEXIT(.T.)     // Einschalten der Tasten
//
@ 10, 10 SAY "Geben Sie ein: " GET cMyVar
READ
READEXIT(lLastExit)            // Alten Zustand wiederherstellen
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

READINSERT() Ein-/Ausschalten des Einf.gemodus f.r READ und MEMOEDIT()

Syntax

READINSERT([<lSchalter>]) -> lSchalterAlt

Parameter

<lSchalter> schaltet den Einf.gemodus ein oder aus. Der Wert true (.T.) schaltet den Einf.gemodus ein, der Wert false (.F.) schaltet ihn aus. Wird der Parameter nicht angegeben, bleibt der Modus unbeeinflu.t.

R.ckgabe

READINSERT() gibt die aktuelle Einstellung als logischen Wert zur.ck.

Anwendung

READINSERT() ist eine Environment-Funktion, die den aktuellen Einf.gemodus f.r READ oder MEMOEDIT() ermittelt. Optional kann der Modus .ber <lSchalter> neu gesetzt werden. Ist der Einf.gemodus ausgeschaltet (READINSERT() liefert false), und gibt der Benutzer Zeichen in ein aktuelles GET-Objekt ein, so .berschreiben diese Zeichen bereits bestehende Zeichen. Ist der Einf.gemodus eingeschaltet (READINSERT() liefert true), werden die Zeichen an der momentanen Cursorposition eingef.gt. Der Einf.gemodus ist ein globaler Wert, der bei Programmstart auf false (.F.) voreingestellt wird und f.r die Laufzeit des Programmes gilt.

READINSERT() kann vor oder w.hrend eines READ oder MEMOEDIT() ausgef.hrt werden. In Kombination mit READ kann READINSERT() aufgerufen werden, wenn die VALID-Bedingung eines GET-Objektes gepr.ft wird, oder innerhalb einer .ber SET KEY definierten Prozedur. In Kombination mit MEMOEDIT() kann READINSERT() innerhalb der Anwenderfunktion oder einer SET KEY-Prozedur ausgef.hrt werden.

Beispiele

Im folgenden Beispiel wird der Einf.gemodus vor einem MEMOEDIT() gesetzt, und nach dem MEMOEDIT() wieder auf den urspr.nglichen Wert zur.ckgestellt:

```
USE Verkauf NEW
lAltModus := READINSERT(.T.)          // Einf.gemodus einschalten
Verkauf->Notizen := MEMOEDIT(Verkauf->Notizen)
READINSERT(lAltModus)                 // Alten Modus herstellen
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

READKEY()* Ermittelt die Taste, mit der ein READ beendet wurde

---

Syntax

READKEY() -> nReadkeyCode

R.ckgabe

READKEY() liefert einen numerischen Wert, der eine Taste repr.sentiert, mit der ein READ verlassen wurde. Die folgende Tabelle enth.lt die Tasten, mit denen ein READ verlassen werden kann, und die von READKEY() gelieferten Werte:

Tabelle 5-27: READKEY() R.ckgabewerte

Taste	R.ckgabewert
Pfeilaufw.rts	5
Pfeilabw.rts	2
Bild.	6
Bild.	7
Strg+Bild.	31
Strg+Bild.	30
Esc	12
Strg+Ende/Strg+W	14
Zeichen nach Ende	15
Return	15

Anwendung

READKEY() ist eine Tastaturfunktion, welche die gleichnamige Funktion aus dBASE III emuliert. Damit kann festgestellt werden, mit welcher Taste ein Benutzer ein READ verlassen hat. Sind Werte w.hrend eines READs modifiziert worden (UPDATED() = .T.), liefert READKEY() den normalen Wert zuz.glich 256. Mit den Tasten [.] und [.] kann ein READ nur dann verlassen werden, wenn READEXIT() auf true (.T.) gesetzt ist.

READKEY() ist nur aus Kompatibilit.tsgr.nden implementiert. Die Funktion kann durch LASTKEY() ersetzt werden. LASTKEY() liefert den Tastencode, der als letztes aus dem Tastaturpuffer geholt wurde. Nach einem READ liefert LASTKEY() den Tastenwert, mit dem das READ verlassen wurde, als Inkey-Code. Um festzustellen, ob irgendein GET-Objekt modifiziert wurde, ist die Funktion UPDATED() anzuwenden.

Dateien:

Bibliothek ist EXTEND.LIB

See Also:

READMODAL() Aktivierung des bildschirmorientierten Editierens einer GET-Liste

---

Syntax

READMODAL(<aGetListe>) -> NIL

Parameter

<aGetListe> ist ein Array, dessen Elemente GET-Objekte enthalten.

R.ckgabe

READMODAL() besitzt keinen R.ckgabewert.

Anwendung

READMODAL() ist eine Benutzerschnittstellen-Funktion, die das Editieren einer Reihe von GET-Objekten erlaubt. READMODAL() ist als GET-READ-System ein Teil der offenen Struktur in Clipper 5.0. Die Funktion ist vergleichbar mit dem READ-Befehl, nimmt aber eine GET-Liste als Parameter an und f.hrt nach dem Verlassen keine Re-Initialisierung durch. Durch dieses Verhalten kann ein mehrfach verschachteltes GET-READ-System realisiert werden.

Um die Kompatibilit.t zu fr.heren Clipper-Versionen aufrecht zu erhalten, ist das GET-System in Clipper 5.0 .ber ein globales Array mit dem Namen GetList implementiert.
Jedesmal, wenn ein @...GET-Befehl bearbeitet wird, wird diesem Array ein neu angelegtes GET-Objekt hinzugef.gt. Der Standard READ-Befehl ist in Clipper 5.0 eine Pre-Prozessor- Anweisung, die mit der Funktion READMODAL() aufgebaut ist und als Parameter das globale Array GetList .bergeben bekommt. Wird keine SAVE-Option angegeben, wird der Variablen GetList ein leeres Array zugewiesen, nachdem die Funktion READMODAL() verlassen wird.

Weitere Informationen .ber das GET-System enth.lt der Abschnitt Get System im Kapitel Grundkonzepte im Referenz-Handbuch I.

Informationen .ber GET-Objekte enth.lt der Abschnitt Die GET-Klasse im Kapitel Objektklassen in diesem Buch.

N.here Informationen .ber die von READMODAL() unterst.tzten Tasten enth.lt die Befehlsbeschreibung von READ im Kapitel Standard-Befehle im Referenz-Handbuch I.

Dateien

Bibliothek ist CLIPPER.LIB, Quell-Datei ist Getsys.prg

See Also:

READVAR() Ermittelt den Namen der aktuellen GET/MENU-Variable

---

Syntax

READVAR() -> cVarName

R.ckgabe

READVAR() liefert den Namen der Variablen eines aktuellen GET-Objektes oder der Variablen, die in einem aktuellen MENU TO-Befehl angegeben ist.

Anwendung

READVAR() ist eine Environment-Funktion und wird in erster Linie bei der Erstellung kontext-sensitiver Hilfesysteme dazu benutzt, das zur Zeit bearbeitete GET-Objekt zu ermitteln. READVAR() arbeitet lediglich w.hrend der Ausf.hrung der READ- und MENU TO-Befehle. Wird die Funktion in anderen wartenden Anweisungen, wie ACCEPT, INPUT, WAIT, ACHOICE(), DBEDIT() oder MEMOEDIT(), benutzt, liefert READVAR eine leere Zeichenkette (""). Sie kann dar.ber hinaus in durch SET KEY angegebenen Prozeduren oder in einer User Defined Function, die .ber eine VALID-Klausel angegeben wurde, angewendet werden.

Beispiele

Das Beispiel veranschaulicht ein einfaches Hilfesystem, in dem in einer Datenbank Hilfe-Informationen .ber mehrere GET-Objekte angegeben sind. Wenn der Benutzer die [F]1-Taste bet.tigt, wird mit READVAR() nach dem Hilfetext gesucht, der zu dem GET-Objekt geh.rt, das der Benutzer gerade bearbeitet:

```
#include "Inkey.ch"
//
SET K_F1 TO Hilfe
cString := SPACE(10)
@ 5, 5 SAY "Eingabe..:" GET cString
READ
RETURN

FUNCTION Hilfe
  USE HelpFile INDEX HelpVar NEW
  SEEK READVAR()
  IF FOUND()
     DispHelp(HelpFile->Text)
  ELSE
     DispHelp(" Keine Hilfe definiert f.r :" + READVAR())
  ENDIF
  CLOSE HelpFile
RETURN NIL

FUNCTION DispHelp( cText )
  LOCAL cScreen := SAVESCREEN(5, 5, 15, 70)
  LOCAL cColor  := SETCOLOR("+BG/B")
  //
  SET CURSOR OFF
  @ 5, 5 CLEAR TO 15, 70
  @ 5, 5 TO 15, 70 DOUBLE
  @ 5, 30 SAY " Hilfe f.r " + READVAR() + " "
  MEMOEDIT(cText, 6, 7, 14, 68, .F.)
  //
  RESTSCREEN(5, 5, 15, 70, cScreen)
  SETCOLOR(cColor)
  SET CURSOR ON
  //
RETURN NIL
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

RECCOUNT()* Ermittelt die Anzahl der Datens.tze in der aktuellen Datenbank

---

Syntax

RECCOUNT()* | LASTREC() -> nDatens.tze

R.ckgabe

RECCOUNT() liefert die Anzahl der physikalischen Datens.tze in der aktuellen Datenbank als ganzzahligen, numerischen Wert. Filterbedingungen, wie SET FILTER oder SET DELETED, haben keinen Einflu. auf den R.ckgabewert. RECCOUNT() liefert Null, wenn keine Datenbank im aktuellen Arbeitsbereich ge.ffnet ist.

Anwendung

RECCOUNT() ist eine Datenbank-Funktion. Sie ist ein Synonym f.r LASTREC() und aus Kompatibilit.tsgr.nden implementiert. RECCOUNT() arbeitet standardm..ig im aktuellen Arbeitsbereich. Durch Voranstellen eines Aliasbezeichners kann die Funktion auch auf Datenbanken in anderen Arbeitsbereichen angewendet werden.

Beispiele

. Das folgende Beispiel zeigt den Unterschied zwischen RECCOUNT() und COUNT:

```
USE Verkauf NEW
? RECCOUNT()                // Resultat: 84
//
SET FILTER TO Verkaeufer = "Schmitz"
COUNT TO nAnzahl
? nAnzahl                   // Resultat: 14
? RECCOUNT()                // Resultat: 84
```

. Dieses Beispiel zeigt, wie die Anzahl von Datens.tzen in Datenbanken, die nicht im aktuellen Arbeitsbereich ge.ffnet sind, ermittelt werden kann:

```
USE Verkauf NEW
USE Kunden NEW
? RECCOUNT()
? Kunden->(RECCOUNT())
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

RECNO() Ermittelt die aktuelle Datensatznummer eines Arbeitsbereiches

---

Syntax

RECNO() -> nDatensatznummer

R.ckgabe

RECNO() gibt die aktuelle Datensatznummer als ganzzahligen, numerischen Wert zur.ck. Enth.lt die im aktuellen Arbeitsbereich ge.ffnete Datenbank keine Datens.tze, liefert RECNO() den Wert 1, EOF() und BOF() ergeben true (.T.), und LASTREC() gibt 0 zur.ck.

Wird der Datensatzzeiger .ber das Ende der Datenbank hinaus bewegt, liefert RECNO() den Wert LASTREC() + 1 und EOF() ist true (.T.). Wird versucht, den Datensatzzeiger vor den ersten Datensatz zu bewegen, liefert RECNO() die Nummer des ersten logischen Datensatzes und BOF() wird true (.T.).

Anwendung

RECNO() ist eine Datenbankfunktion, welche die aktuelle Datensatznummer des aktuellen Arbeitsbereiches liefert. Im Clipper Datenbank-Dateisystem ist jede Datenbank in physikalische Datens.tze eingeteilt. F.r jeden Arbeitsbereich ist ein Zeiger vorgesehen, der auf den aktuellen Datensatz der im jeweiligen Arbeitsbereich ge.ffneten Datenbank zeigt. Dieser Zeiger kann .ber RECNO() ermittelt werden. Durch die Numerierung der Datens.tze wird ein direkter Zugriff auf einzelne Datens.tze erm.glicht, ohne die Datenbank sequentiell zu durchsuchen, um eine bestimmte Position in der Datei zu erreichen.

Eine typische Anwendung f.r RECNO() besteht in Programmteilen, die eine Datenbank satzweise bearbeiten. Dies schlie.t Anweisungen wie SET RELATION ... TO RECNO() ein, die zwei Datenbanken .ber die Datensatznummer miteinander verbindet. Eine weitere Anwendung ist die Anweisung GO RECNO(), die den Inhalt eines Datensatzes im Datei-Puffer aktualisiert.

Voreingestellt arbeitet RECNO() im aktuellen Arbeitsbereich. Um RECNO() auch auf andere Arbeitsbereiche anzuwenden, ist der Funktion ein Aliasbezeichner voranzustellen.

Beispiele

. Das folgende Beispiel fragt nach verschiedenen Datensatzbewegungen .ber RECNO() nach der Datensatznummer:

```
USE Kunden NEW
GO 3
? RECNO()              // Resultat: 3
GO TOP
? RECNO()              // Resultat: 1
nSatz := 5
GO nSatz
? RECNO()              // Resultat: 5
GO BOTTOM
SKIP
? RECNO(), LASTREC()  // Resultat: 11  10
```

. Das Beispiel benutzt Aliasbezeichner, um den Wert von RECNO() in anderen Arbeitsbereichen abzufragen:

```
USE Verkauf NEW
USE Kunden NEW
//
? Verkauf->(RECNO())
? Kunden->(RECNO())
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

RECSIZE() Ermittelt die Datensatzl.nge einer Datenbank

Syntax

RECSIZE() -> nByte

R.ckgabe

RECSIZE() liefert die L.nge eines Datensatzes einer, im aktuellen Arbeitsbereich ge.ffneten Datenbank als ganzzahligen, numerischen Wert. Ist keine Datenbank ge.ffnet, gibt RECSIZE() Null zur.ck.

Anwendung

RECSIZE() ist eine Datenbankfunktion, welche die Datensatzl.nge, durch Addition aller Feldl.ngen, zuz.glich 1 Byte f.r die L.schmarkierung (DELETED()), ermittelt. Dieser Wert, multipliziert mit dem Ergebnis von LASTREC(), liefert den Platzbedarf, den die Datens.tze ben.tigen. Wird zu diesem Ergebnis noch der Wert von HEADER() addiert, erh.lt man den Speicherplatz, den die Datenbank auf der Diskette ben.tigt.

RECSIZE() ist recht hilfreich bei der Erstellung automatischer Backup-Programme. In Verbindung mit DISKSPACE() kann gepr.ft werden, ob gen.gend Speicherplatz auf der Diskette vorhanden ist, bevor eine Datenbank gesichert wird. Bei sehr gro.en Datenbanken, wo zur Sicherung mehrere Disketten ben.tigt werden, l..t sich mit RECSIZE() berechnen, wie viele Datens.tze auf eine Diskette passen.

Als Voreinstellung arbeitet RECSIZE() im aktuellen Arbeitsbereich. Soll die Datensatzl.nge von Datenbanken ermittelt werden, die in anderen Arbeitsbereichen ge.ffnet sind, mu. der Funktion ein Aliasbezeichner vorangestellt werden.

Beispiele

. Die folgende User Defined Function berechnet die Gr..e einer Datei:

```
FUNCTION DbfGroesse
RETURN (RECSIZE() * LASTREC() + HEADER())
```

. Das folgende Beispiel benutzt RECSIZE(), um auf die Satzgr..e in einem nichtselektierten Arbeitsbereich zuzugreifen:

```
USE Kunden NEW
USE Verkauf NEW

? RECSIZE(), Kunden->(RECSIZE())
? DbfGroesse(), Kunden->(DbfGroesse())
```

Dateien:

Bibliothek ist CLIPPER.LIB

See Also:

REPLICATE() Wiederholt eine Zeichenfolge in der angegebenen Anzahl

---

Syntax

REPLICATE(<cZeichenfolge>, <nAnzahl>) -> cZeichenkette

Parameter

<cZeichenfolge> ist die zu wiederholende Zeichenfolge.

<nAnzahl> ist die Anzahl, wie oft <cZeichenfolge> zu wiederholen ist.

R.ckgabe

REPLICATE() gibt eine Zeichenkette zur.ck, die maximal 65.520 Zeichen (64 KB) gro. ist. Wird als Argument f.r <nAnzahl> Null .bergeben, wird eine leere Zeichenkette ("")

Anwendung

REPLICATE() ist eine Zeichenfunktion und wird ben.tigt, wenn gleiche Zeichenfolgen wiederholt auf dem Bildschirm oder Drucker ausgegeben werden m.ssen, oder im Tastaturpuffer abgelegt werden sollen. REPLICATE() ist .hnlich der Funktion SPACE(), bei der Leerzeichen wiederholt werden.

Beispiele

. Diese Beispiele verdeutlichen die R.ckgabe von REPLICATE():

```
? REPLICATE("*", 11)         // Resultat: ***********
? REPLICATE("Hallo ", 2)     // Resultat: Hallo Hallo
? REPLICATE(CHR(42), 11)     // Resultat: ***********
```

. In diesem Beispiel wird eine Reihe von Cursorsteuertasten in den Tastaturpuffer geschrieben:

```
#include "Inkey.ch"
KEYBOARD REPLICATE(CHR(K_DN), 25)
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

RESTSCREEN() Zeigt einen zuvor gesicherten Bildschirmausschnitt an einer angegebenen Stelle

---

Syntax

RESTSCREEN(<nOben>, <nLinks>, <nUnten>, <nRechts>, <cBild>) -> NIL

Parameter

<nOben>, <nLinks>, <nUnten> und <nRechts> sind Bildschirm-Fensterkoordinaten, innerhalb derer die Bildinformationen dargestellt werden sollen.

<cBild> ist die Zeichenkette, die Bildinformationen enth.lt, die angezeigt werden sollen.

R.ckgabe

RESTSCREEN() besitzt keinen R.ckgabewert.

Anwendung

RESTSCREEN() ist eine Bildschirmfunktion und wird benutzt, um Bildschirminformationen, die mit SAVESCREEN() gespeichert worden sind, anzuzeigen. Die Koordinaten sollten gleich zu den Koordinaten sein, mit denen ein Bildschirmausschnitt gesichert wurde, k.nnen aber auch abweichen. Werden andere Koordinaten verwendet, ist darauf zu achten, da. die gleiche Ausschnittgr..e verwendet wird, da es sonst zu unerwarteten Ergebnissen kommen kann. Um mit RESTSCREEN() Bildschirminhalte zu zeigen, die .ber SAVE SCREEN gesichert wurden, m.ssen Sie die Koordinaten mit 0, 0, MAXROW(), MAXCOL() angeben.

Warnung

SAVE SCREEN, RESTORE SCREEN, SAVESCREEN() und RESTSCREEN benutzen den aktuellen Bildschirmtreiber (IBM-PC memory mapped). Werden andere Bildschirmtreiber verwendet, werden diese Funktionen und Befehle u.U. nicht unterst.tzt.

Beispiele

Dieses Beispiel zeigt die Funktion RESTSCREEN() als Teil einer POP_UP-Men.funktion namens PopMenue():

```
? PopMen.({1, 1, 3, 10, {"Auswahl1", "Auswahl2"}, "*BG/B" })

FUNCTION PopMenue( aListe )
  LOCAL cScreen, nAuswahl, cOldColor := SETCOLOR(aListe[6])
  cScreen := SAVESCREEN(aListe[1], aListe[2], aListe[3], aListe[4])
  @ aListe[1], aListe[2] TO aListe[3], aListe[4] DOUBLE
  nAuswahl := ACHOICE(++aListe[1], ++aListe[2],;
    --aListe[3], --aListe[4], aListe[5])
  SETCOLOR(cOldColor)
  RESTSCREEN(--aListe[1], --aListe[2], ++aListe[3],;
    ++aListe[4], cScreen)
  RETURN nAuswahl
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

RIGHT() Liefert den rechten Teil einer Zeichenkette

Syntax

RIGHT(<cZeichenkette>, <nAnzahl>) -> cTeilZeichenkette

Parameter

<cZeichenkette> ist die Zeichenkette, aus der der rechte Teil extrahiert werden soll.

<nAnzahl> ist die Anzahl der Zeichen, die extrahiert werden sollen.

R.ckgabe

RIGHT() gibt <nAnzahl> Zeichen aus <cZeichenkette> von rechts beginnend zur.ck. Ist <nAnzahl> negativ oder Null, wird eine leere Zeichenkette ("") zur.ckgegeben. Ist <nAnzahl> gr..er als die L.nge von <cZeichenkette>, wird die gesamte Zeichenkette zur.ckgegeben. Die maximale L.nge von <cZeichenkette> betr.gt 65.520 (64 KB) Zeichen.

Anwendung

RIGHT() ist eine Zeichenfunktion, die einen Teil einer Zeichenkette, beginnend am rechten Ende, extrahiert. Sie liefert das gleiche Ergebnis wie der Ausdruck: SUBSTR(<cZeichenkette, -<nAnzahl>). Zum Beispiel liefert der Ausdruck RIGHT("ABC",1) das gleiche Ergebnis wie der Ausdruck SUBSTR("ABC",-1). RIGHT() ist .hnlich der Funktion LEFT(), die den linken Teil einer Zeichenkette extrahiert.

Die Funktionen RIGHT(), LEFT() und SUBSTR() werden oft in Kombination mit AT() und RAT() benutzt, um die Position von Zeichenfolgen zu ermitteln, bevor diese extrahiert werden.

Beispiele

. Dieses Beispiel zeigt den Zusammenhang von RIGHT() und SUBSTR():

```
? RIGHT("ABCDEF", 3)         // Resultat: DEF
? SUBSTR("ABCDEF", -3)       // Resultat: DEF
```

. Das folgende Beispiel extrahiert den Teil einer Zeichenkette, der nach dem letzten Auftreten eines Kommas folgt:

```
LOCAL cListe := "Fritz,Peter,Wilhelm"
? RIGHT(cListe, LEN(cListe) -;
   RAT(",", cListe) - 1 )    // Resultat: Wilhelm
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

RLOCK() Sperrt den aktuellen Datensatz im aktiven Arbeitsbereich

---

Syntax

RLOCK() -> lGesperrt

R.ckgabe

RLOCK() liefert true (.T.), wenn die Satzsperrung erfolgreich war, im anderen Fall wird false (.F.) zur.ckgegeben.

Anwendung

RLOCK() ist eine Netzwerkfunktion, die ben.tigt wird, um den aktuellen Datensatz gegen einen Schreibzugriff durch andere, im Netzwerk angemeldete Benutzer zu sch.tzen, bis die Sperrung wieder aufgehoben wird. RLOCK() stellt ein shared lock zur Verf.gung, der anderen Benutzern nur einen Lesezugriff auf den gesperrten Datensatz erm.glicht. .nderungen sind nur dem Benutzer m.glich, der den Datensatz gesperrt hat. Die Sperrung bleibt so lange bestehen, bis ein anderer Datensatz gesperrt wird, eine UNLOCK-Anweisung erfolgt, die Datei geschlossen wird oder eine FLOCK()-Funktion die komplette Datei sperrt.

F.r jeden Aufruf der RLOCK()-Funktion, wird versucht, den aktuellen Datensatz zu sperren und das Ergebnis wird als logischer Wert zur.ckgegeben. Der Versuch, einen Datensatz zu sperren, der von einem anderen Benutzer bereits gesperrt ist oder der die komplette Datei gesperrt hat, schl.gt fehl und die Funktion liefert false (.F.) als R.ckgabewert.

Als Voreinstellung arbeitet RLOCK() im aktuellen Arbeitsbereich. Durch Voranstellen eines Aliasbezeichners kann die Funktion auch auf andere Arbeitsbereiche zugreifen. Diese M.glichkeit ist sehr sinnvoll, da RLOCK() keine automatische Satzsperrung auf Datenbanken, die .ber Relationen miteinander verbunden sind, durchf.hrt.

Generell wird RLOCK() in Netzwerkanwendungen f.r alle Operationen ben.tigt, die den aktuellen Datensatz bearbeiten. Dies umfa.t folgende Befehle:

. @...GET (auf Datenbankfelder)
. DELETE (auf den einzelnen Datensatz)
. RECALL (auf den einzelnen Datensatz)
. REPLACE (auf den einzelnen Datensatz)

Sehen Sie hierzu bitte auch im Kapitel Netzwerkprogrammierung des Buches Programmierung und Hilfsprogramme nach, um weitere Informationen zu erhalten.

Hinweise

. SET RELATION: Clipper sperrt nicht automatisch alle Datens.tze in den mit der aktuellen Datenbank verkn.pften Datenbanken, wenn eine Satzsperrung im aktuellen Arbeitsbereich vorgenommen wird. Gleichfalls hat auch die Anweisung UNLOCK keinen Einflu. auf Relationen.

Beispiele

. Das Beispiel zeigt, wie ein Datensatz in einer Netzwerkumgebung gel.scht werden kann:

```
USE Kunden INDEX Name SHARED NEW
SEEK "Schmitz"
IF FOUND()
  IF RLOCK()
    DELETE
    ? "Schmitz ist gel.scht !"
  ELSE
    ? "Datensatz wird von einem anderen Benutzer ben.tigt !"
  ENDIF
ELSE
  ? "Schmitz wurde nicht gefunden !"
```

```
ENDIF
CLOSE
```

. Das folgende Beispiel zeigt, wie RLOCK() mit einem Aliasbezeichner angewendet wird, um einen Datensatz in einem nicht selektierten Arbeitsbereich zu sperren:

```
USE Verkauf SHARED NEW
USE Kunden SHARED NEW
//
IF .NOT.Verkauf->(RLOCK())
  ? "Der Satz in der Verkaufsdatei wird anderweitig ben.tigt!"
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

ROUND() Rundet einen numerischen Wert auf die angegebenen Dezimalstellen

---

Syntax

ROUND(<nZahl>, <nDezimalstellen>) -> nGerundet

Parameter

<nZahl> ist der zu rundende, numerische Wert.

<nDezimalstellen> ist die Anzahl der Nachkommastellen, auf die gerundet werden soll. Ein negativer Wert rundet auf Vorkommastellen.

R.ckgabe

ROUND() gibt den gerundeten, numerischen Wert zur.ck.

Anwendung

ROUND() ist eine numerische Funktion, die einen in <nZahl> angegebenen Wert auf <nDezimalstellen> rundet. Die Angabe einer Null oder eines negativen Wertes f.r <nDezimalstellen> rundet auf Vorkommastellen. Dezimalzahlen zwischen 5 und 9 werden aufgerundet, Dezimalzahlen unter 5 werden abgerundet.

Die Darstellung des R.ckgabewertes folgt nicht den Einstellungen von DECIMALS, solange SET FIXED auf OFF geschaltet ist. Mit der Einstellung SET FIXED OFF enth.lt die Darstellung des R.ckgabewertes nur so viele Dezimalstellen, wie .ber <nDezimalstellen> angegeben wurde, oder auch keine, wenn <nDezimalstellen> kleiner als Eins ist.

Beispiele

. Das Beispiel zeigt Rundungsergebnisse mit Dezimalstellen:

```
SET DECIMALS TO 2
SET FIXED ON
? ROUND(10.4, 0)           // Resultat: 10.00
? ROUND(10.5, 0)           // Resultat: 11.00
? ROUND(10.51, 0)          // Resultat: 11.00
? ROUND(10.4999999999, 2) // Resultat: 10.50
```

. Das folgende Beispiel benutzt negative Angaben in <nDezimalstellen>:

```
? ROUND(101.99, -1)        // Resultat: 100.00
? ROUND(101.99, -1)        // Resultat: 100.00
? ROUND(109.99, -1)        // Resultat: 110.00
? ROUND(109.99, -2)        // Resultat: 100.00
```

Dateien:

Bibliothek ist CLIPPER.LIB

See Also:

ROW() Ermittelt die aktuelle Bildschirm-Zeilenposition des Cursors

---

Syntax

ROW() -> nZeilennummer

R.ckgabe

ROW() liefert die aktuelle Bildschirmzeile des Cursors als ganzzahligen, numerischen Wert. Der Bereich des R.ckgabewertes liegt zwischen Null und MAXROW().

Anwendung

ROW() ist eine Bildschirmfunktion, welche die aktuelle Position der Zeile ermittelt, in der sich der Cursor befindet. Der Wert in ROW() wird durch Konsolenausgaben (? oder ??), oder direkte Ausgabebefehle und -funktionen aktualisiert. Der Ausgabebefehl @...SAY aktualisiert ROW() nur dann, wenn die aktuelle Einstellung von DEVICE auf SCREEN gesetzt ist.

ROW() kann dazu dienen, den Cursor relativ zur aktuellen Zeilenposition zu bewegen. Die haupts.chliche Anwendung besteht in der Kombination mit COL() und den verschiedenen M.glichkeiten der @... Befehle. COL() und ROW() k.nnen im Einzelfall auch dazu benutzt werden, cursorpositionsabh.ngige Prozeduren oder Funktionen zu entwickeln, um zum Beispiel das aktuelle Eingabefeld nicht zu .berschreiben.

Beispiele

In diesem Beispiel wird ROW() benutzt, um die Anweisung LIST zu simulieren:

```
LOCAL nZeile
USE Kunden INDEX Name NEW
DO WHILE .NOT.EOF()
  CLEAR SCREEN
  @ 1, 1 SAY PADR("Name", LEN(Kunden->Name))
  @ ROW(), COL() + 2 SAY PADR("Adresse", LEN(Kunden->Adresse))
  @ ROW(), COL() + 2 SAY PADR("Telefon", LEN(Kunden->Telefon))
  nZeile := 0
  DO WHILE (nZeile++ <= 23) .AND. (!EOF())
    @ ROW() + 1, 1 SAY Kunden->Name
    @ ROW(), COL() + 2 SAY Kunden->Adresse
    @ ROW(), COL() + 2 SAY Kunden->Telefon
    SKIP
  ENDDO
  WAIT
ENDDO
CLOSE Kunden
```

Dateien:

Bibliothek ist CLIPPER.LIB

See Also:

RTRIM() Entfernt rechtsb.ndige Leerzeichen einer Zeichenkette

---

Syntax

[R]TRIM(<cZeichenkette>) -> cZeichenkette

Parameter

<cZeichenkette> ist die Zeichenkette, in der nachfolgende Leerzeichen zu entfernen sind.

R.ckgabe

RTRIM() liefert eine Kopie von <cZeichenkette>, wobei nachfolgende Leerzeichen entfernt wurden. Ist <cZeichenkette> eine leere Zeichenkette ("") oder enth.lt sie nur Leerzeichen, liefert RTRIM() eine leere Zeichenkette ("").

Anwendung

RTRIM() ist eine Zeichenfunktion, die zur Formatierung von Zeichenketten angewendet wird, wenn rechtsb.ndige Leerzeichen entfernt werden sollen. Bei der Aneinanderreihung von Zeichenketten ist diese Funktion sehr hilfreich, wenn die Reihung ohne Leerzeichen aufeinander folgen soll. Dies ist zum Beispiel bei Datenbankfeldern der Fall, die eine feste L.nge besitzen und dadurch mit Leerzeichen aufgef.llt werden.

RTRIM() ist vergleichbar mit LTRIM(), wo f.hrende Leerzeichen entfernt werden, und ALLTRIM(), wo sowohl f.hrende als auch folgende Leerzeichen entfernt werden. Im Gegensatz zu ALLTRIM(), LTRIM() und RTRIM() f.gen die Funktionen PADC(), PADL() und PADR() Leerzeichen links und/oder rechts an eine Zeichenkette an.

Beispiele

. Das folgende Beispiel ist eine User Defined Function, die RTRIM() benutzt, um L.nderkennzeichen, Postleitzahl und Ort in eine formatierte Zeichenkette zu verwandeln:

```
FUNCTION OrtsAngabe( cLkz, cPlz, cOrt)
RETURN (RTRIM(cLkz) + " " + RTRIM(cPlz) + " " + cOrt )
```

. Das weitere Beispiel zeigt die Anwendung der zuvor aufgef.hrten Funktion Orts-Angabe():

```
USE Kunden INDEX Name NEW
SEEK "Schmitz"
? OrtsAngabe(Lkz, Plz, Ort) // Resultat: D 5000 K.ln
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SAVESCREEN() Speichert einen Bildschirmbereich f.r die sp.tere Anzeige

---

Syntax

SAVESCREEN(<nOben>, <nLinks>, <nUnten>, <nRechts>) -> cBild

Parameter

<nOben>, <nLinks>, <nUnten> und <nRechts> definieren die Koordinaten eines zu sichernden Bildschirmbereiches. .bersteigen die Werte in <nUnten> bzw. <nRechts> die Werte von MAXROW() bzw. MAXCOL(), werden sie auf den zul.ssigen Wert begrenzt.

R.ckgabe

SAVESCREEN() liefert den Inhalt des angegebenen Bildschirmbereiches als Zeichenkette.

Anwendung

SAVESCREEN() ist eine Bildschirmfunktion und wird benutzt, um einen Bereich des Bildschirms in einer Speichervariablen abzulegen. Sp.ter kann der Bildschirminhalt mit der Funktion RESTSCREEN() an der gleichen oder einer anderen Stelle des Bildschirms wieder angezeigt werden.

Warnung

SAVE SCREEN, RESTORE SCREEN, SAVESCREEN() und RESTSCREEN benutzen den aktuellen Bildschirmtreiber (IBM-PC memory mapped). Werden andere Bildschirmtreiber verwendet, werden diese Funktionen und Befehle u.U. nicht unterst.tzt.

Beispiele

Das folgende Beispiel demonstriert den Gebrauch von SAVESCREEN() und RESTSCREEN() in einer PopUp-Men.-Funktion:

```
FUNCTION PopUp(nOben, nLinks, nUnten, nRechts, acAuswahl)
  LOCAL cBild, nAuswahl, cFarbeAlt := SETCOLOR("+BG/B")
  cBild := SAVESCREEN(nOben, nLinks, nUnten, nRechts)
  @ nOben, nLinks TO nUnten, nRechts DOUBLE
  nAuswahl := ACHOICE(++nOben, ++nLinks, --nUnten,;
     --nRechts, acAuswahl)
  RESTSCREEN(--nOben, --nLinks, ++nUnten, ++nRechts, cBild)
  SETCOLOR(cFarbeAlt)
RETURN nAuswahl
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

SCROLL() Rollt einen Bildschirmbereich nach oben oder unten

Syntax

SCROLL(<nOben>, <nLlinks>, <nUnten>, <nRechts>, <nZeilen>) -> NIL

Parameter

<nOben>, <nLlinks>, <nUnten> und <nRechts> definieren die Koordinaten eines Bildschirmbereiches. Zeilen und Spalten k.nnen im Bereich von 0,0 bis MAXROW(), MAXCOL() liegen.

<nZeilen> definiert die Anzahl der Zeilen, um die der Bildschirmbereich nach oben oder unten gerollt werden soll. Werte .ber Null rollen den Bildschirm nach oben, Werte unter Null nach unten. Der Wert Null f.llt den Bildschirmbereich mit Leerzeichen.

R.ckgabe

SCROLL() besitzt keinen R.ckgabewert.

Anwendung

SCROLL() ist eine Bildschirmfunktion, die es erlaubt, einen Bereich des Bildschirms um eine definierte Anzahl von Zeilen nach oben oder unten zu rollen. Wenn der Bereich nach oben gerollt wird, gehen die ersten <nZeilen> verloren, die restlichen Zeilen werden nach oben bewegt und die frei werdenden Zeilen mit Leerzeichen in der aktuell eingestellten Farbe gef.llt. Wird der Bereich nach unten gerollt, ist die Orientierung genau umgekehrt.

SCROLL() wird prim.r dazu benutzt, einen Bildschirmbereich zu verschieben, um zum Beispiel eine Statusinformation in einer neuen Zeile am unteren Bildschirmrand anzuzeigen.

Beispiele

Die folgende, anwenderdefinierte Funktion zeigt am unteren Ende des Bildschirms eine Meldung an, nachdem der Bereich um eine Zeile nach oben verschoben wurde:

```
FUNCTION RollHoch(nOben, nLinks, nUnten, nRechts, expMeldung)
  //
  SCROLL(nOben, nLinks, nUnten, nRechts, 1)
  @ nUnten, nLinks + 1 SAY expMeldung
  //
RETURN NIL
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

SECONDS() Liefert die Sekunden, die seit Mitternacht vergangen sind

---

Syntax

SECONDS() -> nSekunden

R.ckgabe

SECONDS() liefert die Systemzeit als numerischen Wert in der Form von hundertstel Sekunden. Der numerische R.ckgabewert ist die Anzahl der seit Mitternacht verstrichenen Sekunden auf der Basis eines 24-Stundenmodus. Der Wertebereich liegt zwischen 0 und 86399.

Anwendung

SECONDS() ist eine Zeitfunktion, die als einfache Methode zur Berechnung von Zeitspannen verwendet werden kann. Die Basis dieser Berechnung ist die Systemzeit. Die Funktion ist vergleichbar mit der Funktion TIME(), die die Systemzeit in der Form hh:mm:ss als Zeichenkette liefert.

Beispiele

. Dieses Beispiel stellt die Werte von TIME() und SECONDS() gegen.ber:

```
? TIME()       //  Resultat: 10:00:00
? SECONDS()    //  Resultat: 36000.00
```

. Das folgende Beispiel demonstriert den Gebrauch von SECONDS(), um die vergangene Zeit in Sekunden anzuzeigen:

```
LOCAL nStart, nZeitspanne
nStart := SECONDS()
.
.<Anweisungen>...
.
nZeitspanne := SECONDS() - nStart
? "Verstrichene Zeit :", LTRIM(STR(nZeitspanne, 2)), "Sekunden"
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SELECT() Ermittelt die Nummer des aktuellen Arbeitsbereiches

Syntax

SELECT([<cAlias>]) -> nArbeitsbereich

Parameter

<cAlias> ist der Aliasbezeichner des Arbeitsbereiches. Wird der Parameter nicht angegeben, liefert SELECT() die Nummer des aktuellen Arbeitsbereiches.

R.ckgabe

SELECT() liefert die Nummer des spezifizierten Arbeitsbereiches als ganzzahligen, numerischen Wert.

Anwendung

SELECT() ist eine Datenbankfunktion, welche die Nummer eines, .ber den Aliasbezeichner angegebenen Arbeitsbereiches ermittelt. Der zur.ckgegebene Wert kann im Bereich zwischen 0 und 250 liegen. Wird <cAlias> nicht angegeben, gibt SELECT() die Nummer des aktuellen Arbeitsbereiches zur.ck. Wird ein nicht existierender Aliasbezeichner angegeben, liefert SELECT() eine Null.

Hinweise

Die SELECT()-Funktion und der SELECT-Befehl, angegeben mit einem Argument, sind sich .hnlich. Dies sollte kein Problem darstellen, da SELECT() ohne eine Zuweisung innerhalb derselben Zeile keinen Sinn ergibt.

Beispiele

. Das Beispiel zeigt, wie die Anwendung .ber SELECT() feststellen kann, welcher Bereich selektiert wurde:

```
USE Verkauf NEW
SELECT 1
? SELECT("Verkauf") // Resultat: 4
```

. Um tempor.r den Arbeitsbereich zu wechseln, kann mit SELECT() der aktuelle Arbeitsbereich ermittelt und mit dem SELECT-Befehl anschlie.end wie folgt wiederhergestellt werden:

```
LOCAL nOldSelect := SELECT()
USE Verkauf NEW
__.
  . <Anweisungen>
__.
CLOSE
SELECT (nOldSelect)
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET() Pr.ft oder setzt eine globale Einstellung

Syntax

SET(<nKennung>, [<expNeuEinstellung>]) -> AktuelleEinstellung

Parameter

<nKennung> ist ein numerischer Wert, welcher die einzustellende oder zu pr.fende Systemvariable angibt. <nKennung> kann .ber eine symbolische Konstante, die in Set.ch definiert ist, angesprochen werden (siehe weiter unten).

<expNeuEinstellung> ist ein optionaler Parameter, welcher den neu zu setzenden Wert angibt. Der Typ des Parameters h.ngt von der Systemvariablen ab, die .ber <nKennung> angegeben wurde.

Werden ung.ltige Werte in den Parametern <nKennung> und <expNeuEinstellung> angegeben, wird ein Aufruf der SET()-Funktion ignoriert.

R.ckgabe

SET() liefert den aktuellen Wert einer Systemvariablen.

Anwendung

SET() ist eine Systemfunktion, die das Pr.fen und Setzen verschiedener, globaler Clipper-Systemvariablen erm.glicht. Informationen .ber die Bedeutung und die Werte der einzelnen Systemvariablen sind in den SET-Befehlsbeschreibungen aufgef.hrt.

Die zu pr.fenden oder zu setzenden Systemvariablen k.nnen durch symbolische Konstanten angegeben werden. Diese Konstanten sind in der Datei Set.ch definiert. Diese Header-Datei sollte am Anfang eines Programmes eingebunden werden (#include), bevor die Funktion SET() angewendet wird.

In der Header-Datei Set.ch ist ebenfalls eine Konstante _SET_COUNT() definiert. Diese Konstante ist gleich der Anzahl der m.glichen Systemvariablen, die mit SET() gepr.ft oder gesetzt werden k.nnen. Sie erm.glicht die Erstellung einer Funktion, die generell alle Einstellungen von Systemvariablen sichert und neu setzt (siehe folgendes Beispiel).

Hinweis

Der numerische Wert der symbolischen Konstanten, die in Set.ch definiert sind, k.nnen von Version zu Version differieren und sollten daher nie direkt angegeben werden. Benutzen Sie bei Anwendung der SET()-Funktion immer den Namen der jeweiligen Konstanten.

Tabelle 5-28: SET()-Werte definiert in Set.ch

Konstante	Typ	Befehls-/Funktionsbezug
_SET_EXACT	Logisch	SET EXACT
_SET_FIXED	Logisch	SET FIXED
_SET_DECIMALS	Numerisch	SET DECIMALS
_SET_DATEFORMAT	Zeichen	SET DATE
_SET_EPOCH	Numerisch	SET EPOCH
_SET_PATH	Zeichen	SET PATH
_SET_DEFAULT	Zeichen	SET DEFAULT
_SET_EXCLUSIVE	Logisch	SET EXCLUSIVE
_SET_SOFTSEEK	Logisch	SET SOFTSEEK
_SET_UNIQUE	Logisch	SET UNIQUE
_SET_DELETED	Logisch	SET DELETED
_SET_CANCEL	Logisch	SETCANCEL()
_SET_COLOR	Zeichen	SETCOLOR()
_SET_CURSOR	Numerisch	SETCURSOR()
_SET_CONSOLE	Logisch	SET CONSOLE
_SET_ALTERNATE	Logisch	SET ALTERNATE
_SET_ALTFILE	Zeichen	SET ALTERNATE TO
_SET_DEVICE	Zeichen	SET DEVICE

_SET_PRINTER	Logisch	SET PRINTER
_SET_PRINTFILE	Zeichen	SET PRINTER TO
_SET_MARGIN	Numerisch	SET MARGIN
_SET_BELL	Logisch	SET BELL
_SET_CONFIRM	Logisch	SET CONFIRM
_SET_ESCAPE	Logisch	SET ESCAPE
_SET_INSERT	Logisch	READINSERT()
_SET_EXIT	Logisch	READEXIT()
_SET_INTENSITY	Logisch	SET INTENSITY
_SET_SCOREBOARD	Logisch	SET SCOREBOARD
_SET_DELIMITERS	Logisch	SET DELIMITERS
_SET_DELIMCHARS	Zeichen	SET DELIMITERS TO
_SET_WRAP	Logisch	SET WRAP
_SET_MESSAGE	Numerisch	SET MESSAGE
_SET_MCENTER	Logisch	SET MESSAGE

Beispiele

. Das folgende Beispiel zeigt eine User Defined Function, die alle globalen Einstellungen rettet und wiederherstellt. Diese Funktion kann am Anfang und am Ende einer Subroutine stehen, wenn mehrere Einstellungen nur tempor.r ver.ndert werden sollen, ohne die vorherigen Einstellungen zu ver.ndern:

```
#include "Set.ch"
//
FUNCTION SetAll( aNeueEinstellungen )
  LOCAL aAktEinst[_SET_COUNT], n
  IF aNeueEinstellungen <> NIL
    FOR n := 1 TO _SET_COUNT
      aAktEinst[n] := SET(n, aNeueEinstellungen[n])
    NEXT
  ELSE
    FOR n := 1 TO _SET_COUNT
      aAktEinst[n] := SET(n)
    NEXT
  ENDIF
RETURN (aAktEinst)
```

. Das gleiche .ber AEVAL:

```
#include "Set.ch"
//
FUNCTION SetAll( aNewSets )
  LOCAL aOldSets[_SET_COUNT], n := 0
  AEVAL(aOldSets, { ++n, aOldSets[n] := SET(n,;
    IF(aNewSets <> NIL, aNewSets[n],NIL)) } )
RETURN aOldSets
```

Dateien

Bibliothek ist CLIPPER.LIB

SETBLINK() Interpretation des Stern-Symbols (*) bei Farbangaben als "Blink"- oder "heller Hintergrund"-Attribut

---

Syntax

SETBLINK([<lSchalter>]) -> aktuelle Einstellung

Parameter

<lSchalter> schaltet die Bedeutung des Stern-Symbols (*) bei Farbangaben um. Wird der Funktion der logische Wert TRUE (.T.) .bergeben, bedeutet das Stern-Symbol blinkende Darstellung. Wird der logische Wert FALSE (.F.) .bergeben, steht der Stern f.r heller Hintergrund (highlight invers).

R.ckgabe

SETBLINK() liefert die aktuelle Einstellung als logischer Wert.

Anwendung

SETBLINK() ist eine Environment-Funktion, die die Bedeutung des Stern-Symbols zwischen blinken- und hellem Hintergrund umschaltet und die jeweils aktuelle Einstellung zur.ckgibt. Ist SETBLINK() eingeschaltet, werden Farb-Attribute, die an SETCOLOR() .bergeben werden und ein Stern-Symbol enthalten, als Blink-Attribute verstanden. Andernfalls schaltet das Stern-Symbol auf hellen Hintergrund um. Beide Modi k.nnen nicht gleichzeitig aktiv sein.

Hinweis

Diese Funktion ist nur bei IBM-PCs oder kompatiblen Rechnern m.glich, die mit einem EGA- oder VGA- Bildschirmadapter ausger.stet sind.
Beachten Sie ebenfalls, da. die Umschaltung f.r die gesamte Bildschirmdarstellung gilt.

Beispiele

Dieses Beispiel sichert den aktuellen SETBLINK-Status bevor eine Funktion aufgerufen wird, die den SETBLINK-Status ver.ndert. Nach der R.ckkehr wird der alte Status wiederhergestellt:

```
lOldBlink := SETBLINK()
MeinFunk()
SETBLINK( lOldBlin )

FUNCTION MeinFunk
   .
   .
   .
   SETBLINK(.f.)
   SETCOLOR("*W/B")
   .
   .
   .
   RETURN (NIL)
```

See Also:

SETCANCEL() Schaltet den Programmabbruch mit Alt+C ein oder aus

---

Syntax

SETCANCEL([<lSchalter>]) -> lAktuelleEinstellung

Parameter

<lSchalter> schaltet die Funktion von [Alt] + [C] zum Programmabruch ein oder aus. Wird true (.T.) angegeben, kann der Benutzer das Programm mit [Alt]+[C] abbrechen, wird false (.F.) angegeben, wird die Tastenkombination [Alt]+[C] f.r den Programmabbruch gesperrt. Der Vorgabe-wert ist true (.T.).

R.ckgabe

SETCANCEL() gibt die aktuelle Stellung als logischen Wert zur.ck.

Anwendung

SETCANCEL() ist eine Tastaturfunktion und erf.llt zwei Funktionen: Ein- oder Ausschalten der Tastenkombination f.r den Programmabbruch und das Ermitteln der aktuellen Einstellung. Die Funktion wird immer dann benutzt, wenn Sie verhindern wollen, da. ein Programm mit [Alt] + [C] abgebrochen werden kann.

Beachten Sie, da. die Tastaturfunktion SET KEY Vorrang gegen.ber [Alt]+[C] hat, wenn der Schalter auf true (.T.) steht.

Warnung

Ist SETCANCEL() auf false (.F.) gesetzt, kann der Benutzer ein Programm nicht mehr abbrechen, sofern kein anderes Programmende vorgesehen ist.

Beispiele

Dieses Beipiel zeigt einen Weg, ein Programm zu beenden, wenn SETCANCEL() auf false (.F.) steht:

```
#define KEY_ALTC 302
//
SETCANCEL(.F.)                      // Ausschalten von Alt+C
SET KEY KEY_ALTC TO Programmende    // Re-Definition von Alt+C
.
.<Anweisungen>...
.
RETURN

PROCEDURE Programmende
  LOCAL cBild, nAuswahl, cOldColor := SETCOLOR("W/B, N/G")
  SAVE SCREEN TO cBild
  @ 6, 20 CLEAR TO 9, 58
  @ 6, 20 TO 9, 58 DOUBLE
  @ 7; 26 SAY "Wollen Sie das Programm beenden ?"
  @ 8, 35 PROMPT "Ja"
  @ 8, 41 PROMPT "Nein"
  MENU TO nAuswahl
  SETCOLOR(cOldColor)
  RESTORE SCREEN FROM cBild
  //
  IF nAuswahl = 1
    QUIT
  ENDIF
RETURN
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SETCOLOR() Ermittelt und setzt die Werte der aktuellen Farbeinstellung

---

Syntax

SETCOLOR([<cFarbListe>]) -> cFarbListe

Parameter

<cFarbListe> ist eine Zeichenkette, die eine Liste der zu setzenden Farbattribute enth.lt, mit denen Ausgaben auf dem Bildschirm erfolgen. Im folgenden erhalten Sie eine Liste der m.glichen Einstellungen und deren G.ltigkeitsbereiche:

Tabelle 5-29: SETCOLOR() Farbeinstellungen

Attribut	G.ltigkeitsbereich
Standard	Alle Anweisungen und Funktionen, die den Bildschirm beschreiben
Enhanced	GET's und Men.balken
Border	Der Bereich um den Ausgabebereich des Bildschirms. Wird von EGA und VGA nicht unterst.tzt
Background	Nicht unterst.tzt
Unselected	Nicht aktuelle GET-Felder

Die Farbattribute werden in der Form Vordergrund- und Hintergrundfarbe, getrennt durch einen Schr.gstrich "/", angegeben. Die Attribute selbst werden durch Kommata voneinander getrennt. Alle Farbattribute sind optional. Wird ein Attribut .bersprungen, wird der alte Wert beibehalten (siehe auch Beipiele).

R.ckgabe

SETCOLOR() gibt die aktuellen Farbeinstellungen in einer Zeichenkette zur.ck.

Anwendung

SETCOLOR() ist eine Bildschirmfunktion und wird dazu benutzt, die aktuellen Farbeinstellungen zu speichern und optional neue Farbeinstellungen festzulegen. Eine Farbfestlegung besteht aus einer Reihe von Farbangaben, die f.r verschiedene Arten der Bildschirmausgabe benutzt werden. Die einzelnen Farbangaben bestehen jeweils aus einer Vorder- und Hintergrundfarbe, wobei Vordergrund die Farbe der auszugebenden Zeichen und Hintergrund die Farbe hinter den Zeichen bestimmt. Leerzeichen und Zeichen, die keine Ausgabe erzeugen, werden lediglich in der Hintergrundfarbe dargestellt.

In Clipper gibt es mehrere Farbeinstellungen, die f.r die unterschiedlichen Arten der Bildschirmausgabe gelten:

. Standard: Die Farbeinstellung standard wird von allen direkten Bildschirmausgaben und allen Ausgaben auf der Konsole benutzt. Dies gilt f.r die Befehle wie @...PROMPT, @...SAY, ? und f.r die Funktionen wie ACHOICE(), DBEDIT(), MEMOEDIT(), QOUT() usw.

. Enhanced: Diese Einstellung wird f.r extra hervorgehobene Bildschirmanzeigen benutzt. Folgende Anweisungen ber.cksichtigen diese Farbeinstellung: GET's bei SET INTENSITY ON, MENU TO, DBEDIT() und ACHOICE(). Dabei werden nur die aktuellen GET's bzw. die Auswahlbalken hervorgehoben dargestellt.

. Border: Der Rahmen ist der Teil des Bildschirmes, auf dem keine Zeichen dargestellt werden k.nnen. Die Farbeinstellung f.r den Rahmen ist bei manchenBildschirm-Adapterkarten nicht m.glich.

. Background: Diese Farbangabe wird nicht unterst.tzt.

. Unselected: Durch das Farbattribut unselected l..t sich bei Bearbeitung einer Eingabemaske der Blick auf das aktuelle Eingabefeld lenken, indem die restlichen Eingabefelder abgedunkelt werden. Die Farbe f.r diese restlichen Eingabefelder (nicht aktive GET-Objekte) wird durch die Farbangabe unselected bestimmt.

Zus.tzlich k.nnen Farbangaben f.r Vordergrundfarben die Attribute blinkend und intensiv annehmen. Auf einem monochromen Bildschirm erh.ht die intensive Darstellung den Kontrast, auf einem Farbbildschirm .ndert sich die Helligkeit der angegebenen Farbe. So zeigt zum Beispiel N als Vordergrundattribut schwarz, w.hrend +N den Vordergrund als grau darstellt. Eine intensive Darstellung wird mit dem Zeichen "+" gekennzeichnet, w.hrend das Attribut f.r die blinkende Darstellung mit dem Zeichen "*" angegeben wird. Diese beiden Zeichen k.nnen in allen Farbattributen vorkommen, werden aber immer der Vordergrundfarbe zugeordnet.

Folgende Farben werden unterst.tzt:

Tabelle 5-30: SETCOLOR() Farbwert-Tabelle

Farbe	Zeichen	Monochromer Bildschirm
Schwarz	N,Leerzeichen	Schwarz
Blau	B	unterstrichen
Gr.n	G	Wei.
Cyan	BG	Wei.
Rot	R	Wei.
Magenta	RB	Wei.
Braun	GR	Wei.
Wei.	W	Wei.
Grau	N+	Schwarz
Hellblau	B+	Hell unterstrichen
Hellgr.n	G+	Hellwei.
Hellcyan	BG+	Hellwei.
Hellrot	R+	Hellwei.
Hellmagenta	RB+	Hellwei.
Gelb	GR+	Hellwei.
Hellwei.	W+	Hellwei.
Schwarz	U	Unterstrichen
Inverse	I	Inverse Darstellung
Leer	X	Leer

Hinweise

. Nicht angegebene Farbwerte: Im Unterschied zu SET COLOR TO setzt die Funktion SETCOLOR() die Farbwerte nicht auf ihre Standardwerte zur.ck, wenn in der Farbliste <cFarbListe> einzelne Farbangaben .bersprungen bzw. nicht angegeben werden.

. Numerische Farbangaben: Farbwerte in Form von numerischen Werten werden nicht unterst.tzt.

Beispiele

. Dieses Beispiel weist die aktuellen Farbeinstellungen der Variablen cColor zu:

```
cColor := SETCOLOR()
```

. Das n.chste Beispiel zeigt, wie SETCOLOR() angewendet wird, um aktuelle Farbeinstellungen zu sichern und neue Werte zu setzen:

```
cColorNeu := "+RG/B,N/BG,B,,N/W"
cColorAlt := SETCOLOR(cColorNeu)
```

. In dem folgenden Beispiel werden die Farbeinstellungen mit der Anweisung SET COLOR TO auf die Standardwerte zur.ckgesetzt:

```
SET COLOR TO
? SETCOLOR()  //Resultat: W/N, N/W, N, N, N/W
```

. Die n.chsten Beispiele verdeutlichen den Gebrauch von SETCOLOR() mit fehlenden Farbangaben:

```
// Fehlende Angaben am Ende des Parameters
SETCOLOR(("W/N, BG+/B")
```

```
// .berspringen von Farbangaben
SETCOLOR("W/N, BG+/B, , ,W/N")
```

. Das letzte Beispiel zeigt Ihnen die Verwendung von SETCOLOR() zusammen mit ISCOLOR(), um Farbwerte abh.ngig vom verwendeten Bildschirmadapter zu setzen:

```
FUNCTION GrundFarbe( nType )
  STATIC cNormal, cDialog, cFehler
  IF ISCOLOR()
     cNormal := "W/B,+W/GB,,,N/W"
     cDialog := "N/W,W/N,,,N/+N"
     cFehler := "+W/R,+N/W,,,N/W"
  ELSE
     cNormal := "+W/N,N/W,,,W/N"
     cDialog := "+W/N,N/W,,,W/N"
     cFehler := "N/W,+W/N,,,W/N"
  ENDIF
RETURN (SETCOLOR(IF(nType = 0, cNormal,;
       IF(nType = 1, cDialog, cFehler ))))
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SETCURSOR() Ermittelt und setzt die aktuelle Cursorform

Syntax

SETCURSOR([<nCursorForm>]) -> nAktuelleEinstellung

Parameter

<nCursorForm> ist ein numerischer Parameter, der die Form des Cursors angibt. Zur einfacheren Anwendung der Angabe sind in der Header-Datei Setcurs.ch symbolische Konstanten definiert, die die verschiedenen Cursorformen wie folgt beschreiben:

Tabelle 5-31: SETCURSOR() Cursorformen

Form	Wert	Setcur.ch
Kein Cursor	0	SC_NONE
Unterstrich	1	SC_NORMAL
Halber Block unten	2	SC_INSERT
Voller Block	3	SC_SPEZIAL1
Halber Block oben	4	SC_SPEZIAL2

R.ckgabe

SETCURSOR() liefert die aktuelle Cursorform als numerischen Wert zwischen 0 und 4.

Anwendung

SETCURSOR() ist eine Environment-Funktion, welche die Form des Bildschirmcursors kontrolliert. Die aktuelle Form ist abh.ngig vom verwendeten Bildschirmtreiber. Die angegebenen Formen werden nur f.r IBM-PC's und kompatible Computer garantiert. Andere PC-Systeme k.nnen abweichende Ergebnisse liefern.

SETCURSOR(0) hat den gleichen Effekt wie der Befehl SET CURSOR OFF, und SETCURSOR(1) hat die gleiche Wirkung wie SET CURSOR ON.

Beispiele

Das Beispiel zeigt die Verwendung von SETCURSOR(), um auf einen vollen Blockcursor w.hrend eines READs umzuschalten und nach Verlassen des READs den Cursor ganz auszuschalten:

```
#include "Setcurs.ch"
//
USE Kunden NEW
@ 10, 10 GET Kunden->Name
@ 12, 10 GET Kunden->Telefon
//
SETCURSOR(SC_SPECIAL1)
READ
SETCURSOR(SC_NONE)
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist Setcurs.ch

See Also:

SETKEY() Zuweisung eines Aktionsblocks zu einer Taste

Syntax

SETKEY(<nInkeyCode>, [<bAktion>]) -> bAktuelleAktion

Parameter

<nInkeyCode> ist der INKEY()-Wert einer Taste, der einer Aktion zugewiesen oder der abgefragt werden soll.

<bAktion> spezifiziert einen Code-Block, der automatisch dann ausgef.hrt wird, wenn die durch <nInkeyCode> angegebene Taste w.hrend eines Wartezustands bet.tigt wird.

R.ckgabe

SETKEY() liefert den der Taste aktuell zugeordneten Aktionsblock zur.ck, oder NIL, wenn kein Block zugewiesen ist.

Anwendung

SETKEY() ist eine Tastaturfunktion, mit der einem Tastenwert ausf.hrbare Aktionen zu- geordnet werden, oder solche Aktionen erfragt werden sollen. Ist einer Taste eine Aktion zugewiesen, wird diese durchgef.hrt, wenn die entsprechende Taste im Wartezustand bet.tigt wird. Ein Wartezustand liegt immer dann vor, wenn vom Programm ein Tastencode aus dem Tastaturpuffer geholt wird, mit Ausnahme der INKEY()-Funktion. Einbezogen werden die folgenden Funktionen und Befehle: ACHOICE(), DBEDIT, MEMOEDIT, ACCEPT, INPUT, READ und WAIT. Es k.nnen bis zu 32 verschiedene Tasten zur gleichen Zeit einer Aktion zugeordnet sein. Beim Programmstart weist das System die [F1]-Taste automatisch einer Prozedur oder einer User Defined Function mit dem Namen Help zu.

Wenn eine zugewiesene Taste w.hrend eines Wartezustandes bet.tigt wird, werden an die zugewiesene Aktion <bAktion> automatisch drei Parameter, PROCNAME(), PROCLINE() und READVAR(), .bergeben, und die Blockanweisungen in <bAktion> .ber EVAL() ausgef.hrt. Es ist nicht unbedingt notwendig, die .bergebenen Parameter innerhalb der Codeblockdefinition anzugeben, wenn diese nicht ben.tigt werden.

SETKEY() ist .hnlich wie der SET KEY-Befehl, welcher einer Tasten ein Prozeduraufruf zuweist.

Beispiele

Dieser Programmteil zeigt die Zuweisung eines Aktionsblockes zu einer Taste. Anschlie.end wird gezeigt, wie mit EVAL() ein Tastendruck verarbeitet werden kann, wenn INKEY() (kein Wartezustand) zur Tastaturabfrage benutzt wird:

```
#include "Inkey.ch"
SETKEY(K_DN, {|cProc, cLine, cVar| MyProc(cProc, cLine, cVar ) })
.
.<Anweisungen>...
.
DO WHILE LASTKEY() <> K_ESC
  nTaste := INKEY(0)
  IF (bAktion := SETKEY(nTaste)) <> NIL
    EVAL(bAktion, PROCNAME(), PROCLINE(), READVAR())
  ELSEIF nTaste = K_PGUP
    Vor()
  ELSEIF nTaste = K_PGDN
    Zurueck()
  ELSE
    ?? CHR(7)
  ENDIF
ENDDO
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist Inkey.ch

See Also:

SETMODE() Wechselt den Bildschirmmodus auf eine bestimmte Anzahl Zeilen und Spalten.

Syntax

SETMODE(<nZeilen>, <nSpalten>) -> lAusgef.hrt

Parameter

<nZeilen> ist die Anzahl darzustellender Zeilen f.r den Bildschirmmodus.

<nSpalten> ist die Anzahl darzustellender Spalten f.r den Bildschirmmodus.

R.ckgabe

SETMODE() liefert TRUE (.T.), wenn die Umschaltung erfolgt ist, ansonsten wird FALSE (.F.) zur.ckgegeben.

Anwendung

SETMODE() ist eine Environment-Funktion, die versucht, den angeschlossenen Bildschirmadapter auf den entsprechenden Modus zu setzen, der den .bergebenen Zeilen- und Spaltenangaben entspricht. Ein Wechsel des Bildschirmmodus beeinfu.t die entsprechenden Ergebniswerte von MAXROW() und MAXCOL().

Hinweis

Das Umschalten des Bildschirmmodus wird nur von einigen Bildschirmadaptern unterst.tzt.

Beispiele

Das folgende Beispiel schaltet den Bildschirmadapter in den 43-Zeilen-Modus:

```
IF SETMODE(43,80)
  ? "Umschaltung in den 43-Zeilenmodus erfolgreich."
ELSE
  ? "Umschalten in den 43-Zeilenmodus nicht m.glich."
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SETPOS() Bewegt den Cursor an eine neue Position

Syntax

SETPOS(<nZeile>, <nSpalte>) -> NIL

Parameter

<nZeile> und <nSpalte> definieren die neue Bildschirmposition des Cursors. Der Bereich f.r <nZeile> kann zwischen 0 und MAXROW() liegen, und f.r <nSpalte> zwischen 0 und MAXCOL().

R.ckgabe

SETPOS() besitzt keinen R.ckgabewert.

Anwendung

SETPOS() ist eine Bildschirmfunktion und dient dazu, den Cursor an eine neue Bildschirmposition zu bewegen. Nachdem der Cursor gesetzt ist, werden die Werte f.r ROW() und COL() sofort aktualisiert. Um die Form und Sichtbarkeit des Cursors zu ver.ndern, kann SETCURSOR() verwendet werden.

Beispiele

Dieses Beispiel bewegt den Cursor an eine neue Bildschirmposition und gibt eine Zeichenkette .ber den Befehl ?? aus:

```
cString := "Hallo hier bin ich..."
SETPOS(1, 80-LEN(cString))
?? cString
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SETPRC() Setzen der PCOL()- und PROW()-Werte

Syntax

SETPRC(<nZeile>, <nSpalte>) -> NIL

Parameter

<nZeile> ist der neue PROW()-Wert.

<nSpalte> ist der neue PCOL()-Wert.

R.ckgabe

SETPRC() besitzt keinen R.ckgabewert.

Anwendung

SETPRC() ist eine Druckerfunktion. Sie ist sehr hilfreich, wenn Steuersequenzen an einen Drucker geschickt werden sollen, und die Druckkopf-Positionierung mit den Werten in PROW() und PCOL() synchron bleiben soll. Wenn aus einer Clipper-Applikation heraus gedruckt wird, werden die Werte in PCOL() mit der Anzahl der gedruckten Zeichen aktualisiert. Dabei wird nicht unterschieden, ob es sich dabei um druckbare oder nicht druckbare Zeichen handelt. Angenommen, es soll eine 10 Zeichen lange Zeichenfolge an den Drucker geschickt werden und die Zeichenfolge enth.lt zwei Zeichen, die vom Drucker als Steuerzeichen verarbeitet werden, so wird der PCOL()-Wert um 10 erh.ht, w.hrend die Druckkopf-Position sich nur um 8 Spalten verschiebt. Dies kann unter bestimmten Umst.nden zu ungewollten Formatierungsproblemen f.hren. Durch die Anwendung der SETPRC()-Funktion k.nnen Differenzen zwischen PCOL() bzw. PROW() und der tats.chlichen DruckkopfPosition vermieden werden.

SETPRC() kann ebenfalls angewendet werden, wenn bei Druckausgaben .ber @...SAY ungewollte Seitenvorsch.be unterdr.ckt werden sollen. Dies ist dann zweckm..ig, wenn die Zeilenangabe eines @...SAY-Befehls kleiner ist als der aktuelle Wert in PROW(). Normalerweise wird in einem solchen Fall ein Seitenvorschub erzeugt, da der Druckkopf in der Regel nicht r.ckw.rts positioniert werden kann. Sollen diese Effekte unterdr.ckt werden, kann der PROW()-Wert .ber SETPRC() neu gesetzt werden, bevor er in eine Zeile ger.ckt werden soll, die kleiner als PROW() angegeben ist.

Beispiele

Die folgende User Defined Function PrintCodes() zeigt, wie SETPRC() angewendet wird, um Steuerzeichen an einen Drucker zu senden, ohne die Werte von PROW() und PCOL() zu ver.ndern:

```
#include "Set.ch"
#define ITALIC_ON CHR(27) + "I"
#define ITALIC_OFF  CHR(27) + "E"
//
SET DEVICE TO PRINTER
@ 12, 10 SAY " Dies ist ein"
@ PROW(), PCOL() + 2 SAY PrintCodes(ITALIC_ON) + "wichtiges"
@ PROW(), PCOL() + 2 SAY PrintCodes(ITALIC_OFF) + "treffen"
SET DEVICE TO SCREEN

FUNCTION PrintCodes(cSteuerzeichen)
  LOCAL nZeile := PROW(), nSpalte := PCOL(), lDrucker
  lDrucker := SET(_SET_PRINTER, .T.)
  //
  ?? cSteuerzeichen
  //
  SETPRC(nZeile, nSpalte)
  SET(_SET_PRINTER, lDrucker)
RETURN ""
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SOUNDEX() Konvertiert eine Zeichenkette in ihre Lautform

Syntax

SOUNDEX(<cZeichenkette>) -> cLautform

Parameter

<cZeichenkette> ist die zu konvertierende Zeichenkette.

R.ckgabe

SOUNDEX() liefert eine vierstellige Zeichenkette in der Form A999.

Anwendung

SOUNDEX() ist eine Zeichenfunktion, die in Indexschl.sseln benutzt wird, wenn nach .hnlich klingenden Worten gesucht werden soll. Sie wird in Anwendungen ben.tigt, wo nach Namen oder Begriffen gesucht werden soll, deren exakte Schreibweise nicht bekannt ist. SOUNDEX() wird vor allem in solchen Anwendungen benutzt, in denen nach Namen von Personen gesucht wird. Gerade bei der Eingabe von Namen treten immer wieder Fehler auf (z.B. Meier, Maier oder Mayer), sei es, da. der Name durch Unkenntnis falsch geschrieben oder telefonisch nur ungenau durchgegeben wird. Soundex() macht aus gleichlautenden Ausdr.cken einen gleichen Indexschl.ssel und sorgt auf diese Weise daf.r, da. ein Name auch dann gefunden wird, wenn die Schreibweise nicht .bereinstimmt.

Beispiele

. Dieses Beispiel erzeugt einen Index mit der Funktion SOUNDEX() und sucht dann nach einem Namen:

```
USE Kunden
INDEX ON SOUNDEX(Name) TO KundName
//
SEEK SOUNDEX("Meier")
? FOUND(), Kunden->Name //Resultat: .T. Meier
```

. Danach wird nach dem gleichen Namen in anderer Schreibweise gesucht:

```
SEEK SOUNDEX("Maier")
? FOUND(), Kunden->Name //Resultat: .T. Meier
```

Dateien

Bibliothek ist EXTEND.LIB, Quelldatei ist EXAMPLEC.C

See Also:

SPACE() Erzeugt eine Zeichenkette mit Leerzeichen

Syntax

SPACE(<nAnzahl>) -> cZeichenkette

Parameter

<nAnzahl> ist die Anzahl von Leerzeichen. Der Wert darf maximal 65.520 (64 KB) Zeichen betragen.

R.ckgabe

SPACE() gibt eine Zeichenkette mit <nAnzahl> Leerzeichen zur.ck. Wird als Argument Null .bergeben, besteht der R.ckgabewert aus einer leeren Zeichenkette ("").

Anwendung

SPACE() ist eine Zeichenfunktionen und wird benutzt, um eine Folge von Leerzeichen zu erzeugen. Sie liefert das gleiche Ergebnis wie der Ausdruck REPLICATE(" ",<nAnzahl>). SPACE() wird oft dazu benutzt, Eingabevariablen zu initialisieren, die sp.ter mit @...GET editiert werden sollen. SPACE() dient also zum F.llen einer Zeichenkette mit Leerzeichen.

Beispiele

Dieses Beispiel benutzt SPACE() f.r die Initialisierung von Eingabefeldern:

```
USE Kunden NEW
MEMVAR->Name := SPACE(LEN(Kunden->Name))
@ 10, 10 SAY "Name des Kunden: " GET MEMVAR->Name
READ
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SQRT() Errechnet die Wurzel einer positiven Zahl

Syntax

SQRT(<nZahl>) -> nWurzel

Parameter

<nZahl> ist die Zahl, deren Wurzel errechnet werden soll.

R.ckgabe

SQRT() gibt einen doppelt genauen, numerischen Wert zur.ck. Die Anzahl der angezeigten Nachkommastellen h.ngt von der Stellung der Anweisung SET DECIMALS ab, SET FIXED wird nicht ber.cksichtigt. Ein negativer Wert in <nZahl> erzeugt den Wert Null.

Anwendung

SQRT() ist eine numerische Funktion und wird in jedem beliebigen numerischen Ausdruck benutzt, um die Wurzel zu berechnen. Zum Beispiel kann sie in einem Ausdruck verwendet werden, um die Standardabweichung einer Reihe von Zahlen zu berechnen.

Beispiele

Das Beispiel zeigt die verschiedenen Ergebnisse von SQRT():

```
SET DECIMALS TO 5
//
? SQRT(2)          // Resultat: 1.41421
? SQRT(4)          // Resultat: 2.00000
? SQRT(4) ** 2     // Resultat: 4.00000
? SQRT(2) ** 2     // Resultat: 2.00000
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

STR() Konvertiert einen numerischen Ausdruck in eine Zeichenkette

---

Syntax

STR(<nZahl>, [<nL.nge>], [<nDezimalstellen>]) -> cZahl

Parameter

<nZahl> ist ein numerischer Wert, der in eine Zeichenkette konvertiert werden soll.

<nL.nge> ist die L.nge der zur.ckzugebenden Zeichenkette einschlie.lich Nachkommastellen, Vorzeichen und dem Komma selbst.

<nDezimalstellen> ist die Anzahl der Nachkommastellen.

R.ckgabe

STR() gibt <nZahl> als formatierte Zeichenkette zur.ck. Werden <nL.nge> oder <nDezimalstellen> nicht angegeben, verf.hrt Clipper nach folgenden Regeln:

Tabelle 5-32: STR() Ergebnisse von STR() ohen opt. Param.

Ausdruck	L.nge der R.ckgabe
Feld-Variable	Feldl.nge + Dezimalstellen
Ausdr.cke/Konstanten	Min. 10 Zeichen + Dezimalstellen
VAL()	Min. 3 Zeichen
MONTH()/DAY()	3 Zeichen
YEAR()	5 Zeichen
RECNO()	7 Zeichen

Anwendung

STR() ist eine numerische Konvertierungsfunktion, die einen numerischen Wert in eine Zeichenkette wandelt. Sie wird in der Regel dazu benutzt, numerische Werte mit Zeichenketten zu verbinden. STR() wird in Applikationen zur Darstellung numerischer Werte, Erzeugung von Schl.sseln mit numerischen Werten und zur Bildung von Indexausdr.cken in Verbindung mit Zeichenketten verwendet.

STR() ist .hnlich der Funktion TRANSFORM(), die die Konvertierung eines numerischen Wertes in eine Zeichenkette .ber eine Formatmaske vornimmt. Da TRANSFORM() eine Maske verwendet, k.nnen weitere Zeichen, wie W.hrungssymbole, Klammern oder Trennzeichen bei der Konvertierung eingebracht werden.

Die inverse Funktion zu STR() ist VAL(). Sie konvertiert einen numerischen Zeichenkettenausdruck in einen numerischen Wert.

Hinweise

. Ist die Angabe in <nL.nge> kleiner als die Anzahl der darzustellenden Zahlen vor dem Komma, wird eine Zeichenkette mit Sternen (*) anstatt Zahlen zur.ckgegeben.

. Ist die Angabe in <nL.nge> kleiner als der zur Darstellung der Nachkommastellen ben.tigte Wert, wird der zur.ckgegebene Wert gerundet.

. Wird <nL.nge> angegeben und der Parameter <nDezimalstellen> weggelassen, so wird der R.ckgabewert zu einem ganzzahligen Wert gerundet.

Beispiele

. Das Beispiel zeigt die R.ckgabewerte von STR() bei .bergabe unterschiedlicher Werte:

```
nZahl := 123.45
? STR(nZahl)            // Resultat: 123.45
? STR(nZahl, 4)         // Resultat: 123
? STR(nZahl, 2)         // Resultat:        **
? STR(nZahl*10, 7, 2)   // Resultat: 1234.50
? STR(nZahl*10, 12, 4)  // Resultat: 1234.5000
```

```
? STR(nZahl, 10, 1)      // Resultat: 1234.5
```

. Das folgende Beispiel benutzt STR(), um einen Indexschl.ssel,
bestehend aus einer Zahl und einer Zeichenkette, zu bilden:

```
USE Kunden NEW
INDEX ON STR(Umsatz / 1000, 6) + Name TO Umsatz
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

STRTRAN() Sucht und ersetzt Zeichen innerhalb einer Zeichenkette

---

Syntax

```
STRTRAN(<cZeichenkette>, <cSuchfolge>,
       [<cErsatzfolge>],
       [<nStart>],
       [<nAnzahl>]) -> cZeichenketteNeu
```

Parameter

<cZeichenkette> ist die zu bearbeitende Zeichenkette oder ein Memofeld.

<cSuchfolge> sind die zu suchenden Zeichen.

<cErsatzfolge> sind die Zeichen, mit denen ersetzt wird. Wird dieses Argument nicht angegeben, wird mit einer leeren Zeichenkette ("") ersetzt.

<nStart> bezeichnet die erste Position, ab der ersetzt wird. Der Vorgabewert betr.gt Eins.

<nAnzahl> ist die Anzahl der Ersetzungen, die vorgenommen werden. Als Vorgabe werden alle Zeichen bis zum Ende von <cZeichenkette> ersetzt.

R.ckgabe

STRTRAN() gibt eine Zeichenkette zur.ck, in der <cSuchfolge> durch <cErsatzfolge> ersetzt worden ist.

Anwendung

STRTRAN() ist eine Zeichenfunktion, die eine Suche nach einer Zeichenfolge in der angegebenen Zeichenkette durchf.hrt. Wird eine .bereinstimmung gefunden, ersetzt STRTRAN() das Vorkommen von <cSuchzeichen> gegen <cErsatzfolge>. Werden <nStart> und <nAnzahl> nicht angegeben, werden alle Vorkommen bis zum Ende von <cZeichenkette> ersetzt. Beachten Sie, da. STRTRAN() Teilzeichenketten ersetzt, und deshalb nicht auf das Vorkommen von ganzen Worten geachtet wird.

Beispiele

Das folgende Beispiel benutzt STRTRAN(), um eine Ersetzung vorzunehmen:

```
cString := "compilieren, oder nicht compilieren?"
? STRTRAN(cString, "compilieren", "sein")     //"sein, oder nicht sein?"
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

STUFF() L.schen und Einf.gen von Zeichen in einer Zeichenkette

---

Syntax

STUFF(<cZeichenkette>, <nStart>, <nL.schen>, <cEinf.gung>) -> cZeichenketteNeu

Parameter

<cZeichenkette> ist die Zeichenkette, in der Zeichen gel.scht und eingef.gt werden sollen.

<nStart> bezeichnet die Position, ab der gel.scht bzw. eingef.gt wird.

<nL.schen> ist die Anzahl der zu l.schenden Zeichen.

<cEinf.gung> ist die Zeichenkette, die eingef.gt wird.

R.ckgabe

STUFF() gibt eine Zeichenkette zur.ck, in der die entsprechende Anzahl von Zeichen gel.scht und eine Zeichenkette eingef.gt worden ist.

Anwendung

STUFF() ist eine Zeichenfunktion. Sie l.scht <nL.schen> Zeichen beginnend bei <nStart> und f.gt dann ab <nStart> eine Zeichenkette <cEinf.gung> ein. Aus dieser M.glichkeit resultieren die folgenden sechs Operationen mit STUFF():

. Einf.gen: Ist <nL.schen> gleich Null, werden keine Zeichen gel.scht. Dadurch wird lediglich <cEinf.gung> ab <nStart> in die Zeichenkette aufgenommen. So ergibt der Ausdruck STUFF("Auf meinem Hund sitzen Fliegen", 24, 0, "keine ") die Zeichenkette "Auf meinem Hund sitzen keine Fliegen".

. Ersetzen: Hat <cEinf.gung> dieselbe L.nge wie <nL.schen>, wird ab <nStart> ersetzt. Es wird also die gleiche Anzahl an Zeichen gel.scht, wie eingef.gt wurde, wodurch die zur.ckgegebene Zeichenkette eine unver.nderte L.nge besitzt. So ergibt der Ausdruck STUFF("Mein Hund hat Fl.he", 15, 5, "B.lle") die Zeichenkette "Mein Hund hat B.lle".

. L.schen: Ist <cEinf.gung> eine leere Zeichenkette (""), wird lediglich die in <nL.schen> angegebene Anzahl von Zeichen entfernt. Der Ausdruck STUFF("Auf meinem Hund sitzen keine Fliegen", 24, 6, "") ergibt die Zeichenkette "Auf meinem Hund sitzen Fliegen".

. Ersetzen und Einf.gen: Ist die Zeichenfolge in <cEinf.gung> l.nger als <nL.schen>, werden zun.chst alle Zeichen bis zum Ende von L.schung ersetzt und dann der Rest aus <cEinf.gung> in die Zeichenkette aufgenommen. Die resultierende Zeichenkette ist also l.nger als <cZeichenkette>. Der Ausdruck STUFF("Mein Hund hat Fl.he", 11, 3, "hatte niemals") erzeugt die Zeichenkette "Mein Hund hatte niemals Fl.he".

. Ersetzen und L.schen: Ist die L.nge von <cEinf.gung> kleiner als <nL.schen>, werden mehr Zeichen entfernt, als eingef.gt. Die resultierende Zeichenkette ist also k.rzer als <cZeichenkette>. Der Ausdruck STUFF("Mein Hund hatte niemals Fl.he", 11, 13, "fri.t") ergibt die Zeichenkette "Mein Hund fri.t Fl.he".

. Ersetzen und Rest l.schen: Ist <nL.schen> gr..er oder gleich der Anzahl von Zeichen ab <nStart> bis zum Ende der Zeichenkette, werden zun.chst alle Zeichen bis zum Ende der Zeichenkette gel.scht, bevor eingef.gt wird. Der Ausdruck STUFF("Mein Hund hat Fl.he", 11, 9, "fri.t") erzeugt die Zeichenkette "Mein Hund fri.t".

Beispiele

Die folgenden sechs Beispiele sollen Ihnen noch einmal die M.glichkeiten von STUFF() verdeutlichen:

```
//Einf.gen
? STUFF("ABCDEF", 2, 0, "xyz")    // Resultat: AxyzBCDEF
```

```
//Ersetzen
? STUFF("ABCDEF",2, 3, "xyz)       // Resultat: AxyzEF

//L.schen
? STUFF("ABCDEF", 2, 2, "")        // Resultat: ADEF

// Ersetzen und Einf.gen
? STUFF("ABCDEF", 2, 1, "xyz")     // Resultat: AxyxCDEF

//Ersetzen und L.schen
? STUFF("ABCDEF", 2, 4, "xyz")     // Resultat: AxyzF

//Ersetzen und Rest l.schen
? STUFF("ABCDEF", 2, 10, "xyz")    // Resultat: Axyz
```

Dateien

Bibliothek ist CLIPPER.LIB, Quelldatei ist EXAMPLEC.C

See Also:

SUBSTR() Extrahieren einer Zeichenfolge aus einer Zeichenkette

---

Syntax

SUBSTR(<cZeichenkette>, <nStart>, [<nAnzahl>]) -> cTeilzeichenkette

Parameter

<cZeichenkette> ist die Zeichenkette, aus der ein Teil herausgezogen werden soll. Die maximale Gr..e von <cZeichenkette> betr.gt 65.520 Zeichen (64 KB).

<nStart> ist die Anfangsposition in <cZeichenkette>. Ist <nStart> positiv, bezeichnet <nStart> das erste Zeichen der Teilzeichenkette. Ist <nStart> negativ, bezeichnet <nStart> die letzten Zeichen der Teilzeichenkette.

<nAnzahl> ist die L.nge der Teilzeichenkette. Wird <nAnzahl> nicht angegeben, wird die Teilzeichenkette bis zum Ende von <cZeichenkette> ermittelt. Ist <nAnzahl> gr..er, als die von <nStart> zur Verf.gung stehende Anzahl von Zeichen, wird die .bersch.ssige L.nge ignoriert.

R.ckgabe

SUBSTR() gibt eine Zeichenkette zur.ck.

Anwendung

SUBSTR() ist eine Zeichenfunktion und liefert einen bestimmten Teil einer Zeichenkette oder eines Memofeldes. SUBSTR() ist vergleichbar mit den Funktionen LEFT() und RIGHT(), die Zeichenfolgen, vom linken oder rechten Rand beginnend, liefern.

Die Funktionen SUBSTR(), LEFT() und RIGHT() werden oft in Kombination mit AT() und RAT() angewendet, um eine Zeichenfolge zu lokalisieren, bevor sie extrahiert wird. Ebenso werden sie ben.tigt, wenn Teile einer Zeichenkette auf dem Bildschirm oder Drucker ausgegeben werden sollen.

Beispiele

. Das folgende Beispiel ermittelt den Vor- und den Nachnamen aus einer Zeichenkette:

```
cName := "Harald Schmidt"
? SUBSTR(cName, 1, 6)           // Resultat: Harald
? SUBSTR(cName, 7, 7)           // Resultat: Schmidt
? SUBSTR(cName, LEN(cName)+ 2)) // Resultat: ""
? SUBSTR(cName, -7)             // Resultat: Schmidt
? SUBSTR(cName, -7, 3)          // Resultat: Sch
```

. Dies Beispiel zeigt die Anwendung von SUBSTR() in Kombination mit AT() und RAT(), um damit eine User Defined Function zu definieren, die den Dateinamen aus einer Dateispezifikation ermittelt:

```
? FileBase("C:\PRG\TEST.OBJ") // Resultat: TEST.OBJ

FUNCTION FileBase( cDatei )
  LOCAL nPos
  IF (nPos := RAT("\", cDatei)) <> 0
    RETURN SUBSTR(cDatei, nPos + 1)
  ELSEIF (nPos := AT(":", cDatei)) <> 0
    RETURN SUBSTR(cDatei, nPos + 1)
  ELSE
    RETURN cDatei
  ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

TIME() Ermittelt die Systemzeit

Syntax

TIME() -> cSystemzeit

R.ckgabe

TIME() gibt die Systemzeit als Zeichenkette in der Form hh:mm:ss zur.ck, wobei ss die Stunden, mm die Minuten und ss die Sekunden angibt. Stunden, Minuten und Sekunden werden durch einen Doppelpunkt getrennt.

Anwendung

TIME() ist eine Zeitfunktion und wird zur Zeitdarstellung auf dem Bildschirm oder Drucker ben.tigt. TIME() ist vergleichbar mit der Funktion SECONDS(), welche die Zeit als Zahlenwert in Sekunden seit Mitternacht liefert. SECONDS() l..t sich daher besser bei Berechnungen einsetzen.

Beispiele

. Das folgende Beispiel zeigt die verschiedenen Ergebnisse von TIME() in Verbindung mit SUBSTR(), um Stunden, Minuten und Sekunden zu separieren:

```
? TIME()                  // Resultat: 10:37:17
? SUBSTR(TIME(), 1, 2)  // Resultat: 10
? SUBSTR(TIME(), 4, 2)  // Resultat: 37
? SUBSTR(TIME(), 7, 2)  // Resultat: 17
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

TONE() Erzeugt einen Ton in angegebener Frequenz und Dauer

Syntax

TONE(<nFrequenz>, <nDauer>) -> NIL

Parameter

<nFrequenz> ist ein positiver numerischer Wert, der die Frequenz des Tons spezifiziert.

<nDauer> ist ein positiver numerischer Wert, der die Dauer des Tones in 1/18 Sekunden angibt. Ein Ton mit einer Sekunde Dauer ben.tigt also den Wert 18.

Beachten Sie bitte, da. bei beiden Parametern die Nachkommastellen abgeschnitten und nicht gerundet werden.

R.ckgabe

TONE() besitzt keinen R.ckgabewert.

Anwendung

TONE() ist eine Sound-Funktion, die benutzt werden kann, um einem Benutzer verschiedene Status.nderungen eines Programmes mitzuteilen. Dies ist m.glich bei der Anzeige einer Fehlernachricht, oder wenn ein Grenzwert .ber- oder unterschritten wird. Durch mehrfachen Aufruf dieser Funktion mit unterschiedlichen Angaben kann f.r jeden Status eine eigene Tonfolge erzeugt werden. So kann zum Beispiel beim Auftreten eines Fehlers eine definierte Dreitonfolge ert.nen, bevor ein Dialogfenster ge.ffnet wird. Wenn in einem TBrowse-Objekt versucht wird, den Datensatzzeiger vor den Beginn oder hinter das Ende einer Datenbank zu stellen, ist das Ert.nen einer Tonfolge f.r den Benutzer hilfreich, um ihm den jeweiligen Status mitzuteilen.

TONE() arbeitet .ber den PC-internen Lautsprecher, um einen Ton mit spezifizierter Dauer ert.nen zu lassen. Die Dauer wird dabei in Schritten zu 18tel Sekunden angegeben. Die Frequenz wird in Hertz (Perioden pro Sekunde) angegeben. Frequenzen unter 20 Hertz sind nicht h.rbar und daher zu vermeiden. Die weiter unten aufgef.hrte Tabelle zeigt die Frequenzen der musikalischen Noten.

Tabelle 5-33: TONE() Frequenzen der Musiknoten

Note	Frequenz	Note	Frequenz
C	130.80	'C	261.70
C#	138.60	'C#	277.20
D	146.80	'D	293.70
D#	155.60	'D#	311.10
E	164.80	'E	329.60
F	174.60	'F	349.20
F#	185.00	'F#	370.00
G	196.00	'G	392.00
G#	207.70	'G#	415.30
A	220.00	'A	440.00
A#	233.10	'A#	466.20
B	246.90	'B	493.90
		"C	523.30

Hinweis

TONE() funktioniert ausschlie.lich auf IBM-PC's und 100% kompatiblen Systemen.

Beispiele

. Das folgende Beispiel zeigt eine Tonfolge-Funktion, die Sie dazu benutzen k.nnen, das Ende einer Operation anzuzeigen:

```
FUNCTION BeepFertig
  TONE(300, 1)
  TONE(100, 1)
  TONE(300, 1)
  TONE(100, 1)
```

```
RETURN NIL
```

. Dieses Beispiel zeigt eine Tonfolge, welche die Fehleingabe anzeigt:

```
FUNCTION BeepFehler
  TONE(100, 3)
RETURN NIL
```

Dateien

Bibliothek ist CLIPPER.LIB, Quelldatei ist EXAMPLEA.ASM

See Also:

TRANSFORM() Konvertiert einen beliebigen Wert in eine formatierte Zeichenkette

---

Syntax

TRANSFORM(<exp>, <cMaske>) -> cZeichenkette

Argumente:

<exp> ist der zu formatierende Wert. Der Ausdruck kann ein beliebiger Clipper-Datentyp mit Ausnahme von Array und NIL sein.

<cMaske> ist eine Zeichenfolge bestehend aus Schablonen und Formatierungszeichen, welche die Form der zur.ckgegebenen Zeichenkette beschreibt.

R.ckgabe

TRANSFORM() gibt eine durch <cMaske> beschriebene, formatierte Zeichenkette zur.ck.

Anwendung

TRANSFORM() ist eine Konvertierungsfunktion, die es erm.glicht, einen Zeichenausdruck, ein Datum, einen logischen oder einen numerischen Wert in eine, durch eine Maske angegebene, formatierte Zeichenkette umzuwandeln. Die Umwandlung erfolgt dabei in der gleichen Weise, wie die Anweisung PICTURE in einem @...SAY-Befehl. TRANSFORM() wird in der Regel zur Darstellung von Daten auf dem Bildschirm oder Drucker ben.tigt.

. Formatierungs-Funktionen: Eine Formatierungsfunktion gibt eine Regel an, nach welcher TRANSFORM() die Umwandlung von <exp> vornehmen soll. Die Angabe einer Formatierungsfunktion erfolgt in einer Zeichenkette und beginnt mit einem Klammeraffen (@), gefolgt von einem oder mehreren Zeichen. Jedes Zeichen steht f.r eine bestimmte Bedeutung, die in der folgenden Liste aufgef.hrt ist. Die Zeichenkette <cMaske> mu. zur Angabe einer Formatierungsfunktion als erstes Zeichen einen Klammeraffen enthalten und es d.rfen keine Leerzeichen vorkommen. Die Angabe einer Formatierungsfunktion kann alleine oder zusammen mit Schablonen angegeben werden. Wird beides angegeben, mu. die Formatierfunktion von den Schablonen durch ein einzelnes Leerzeichen getrennt werden.

Tabelle 5-34: TRANSFORM() Formatierungsfunktionen

Funktion	Wirkung
B	Linksb.ndige Anzeige von Zahlen
C	Anzeige von CR nach positiven Zahlen (Credit)
D	Zeigt das Datum im Format von SET DATE
E	Zeigt das Datum im englischen Format
R	Zeichen, die keine Schablonen sind, werden eingef.gt
X	Zeigt DB hinter negativen Zahlen an (Debit)
Z	Zeigt Nullen anstatt f.hrende Leerzeichen
(	Schlie.t negative Zahlen in Klammern ein
!	Konvertiert nach Gro.buchstaben

. Schablonen: Schablonen spezifizieren die Formatierung einzelner Zeichen. Die Formatierungsanweisung .ber Schablonen enth.lt eine Reihe von Zeichen, die, wie in der unten aufgef.hrten Tabelle gezeigt, Darstellungen einzelner Zeichen bestimmen. Jede Position in der Schablonenfolge repr.sentiert eine Position in der zu formatierenden Zeichenkette. Zeichen, die keine Formatieraufgabe haben, erscheinen so in der formatierten Zeichenkette, wie sie angegeben wurden. Wenn gleichzeitig die Formatierfunktion @R verwendet wird, werden diese Zeichen zwischen den einzelnen Zeichen des Ausgabewertes eingef.gt. Andernfalls .berschreiben solche Zeichen die entsprechenden Stellen. Schablonenangaben k.nnen einzeln, oder gemeinsam mit einer Formatierfunktion angegeben werden. Werden beide Arten verwendet, mu. die Formatierfunktion zuerst angegeben und von den Schablonen durch ein Leerzeichen getrennt werden.

Tabelle 5-35: TRANSFORM() Schablonen

Platzhalter	Wirkung

A, N, X, 9, #	Zeigt Ziffern f.r jeden Datentyp
L	Zeigt logische Werte als "T" oder "F"
Y	Zeigt logische Werte als "J" oder "N"
!	Wandelt Klein- in Gro.buchstaben um
$	Zeigt das $-Symbol anstelle von Leerzeichen
*	Zeigt das *-Symbol anstelle von Leerzeichen
.	Gibt die Position des Dezimalpunktes an
,	Gibt die Position des Kommas an

Beispiele

. Das folgende Beispiel konvertiert eine Zahl in ein W.hrungsformat unter Benutzung von Schablonen:

```
? TRANSFORM(123456, "$999,999.99")  // Resultat: $123,456.00
```

. Im folgenden Beispiel wird eine Zeichenkette .ber eine Formatierfunktion formatiert:

```
? TRANSFORM("klein", "@!")                  // Resultat: KLEIN
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

TYPE() Ermittelt den Datentyp eines Ausdrucks

Syntax

TYPE(<cAusdruck>) -> cTyp

Parameter

<cAusdruck> ist der Ausdruck, angegeben als Zeichenkette, dessen Typ bestimmt werden soll. <cAusdruck> kann dabei ein Feldname mit oder ohne Aliasbezeichner, eine Speichervariable oder ein Ausdruck beliebigen Typs sein.

R.ckgabe

TYPE() gibt eine der folgenden Zeichen zur.ck:

Tabelle 5-36: TYPE() R.ckgabewerte

R.ckgabewert	Bedeutung
A	Array
B	Block
C	Zeichen
D	Datum
L	Logisch
M	Memo
N	Numerisch
U	NIL, LOCAL oder STATIC
UE	Syntaxfehler
UI	Unbestimmter Ausdruck

Anwendung

TYPE() ist eine Clipper-Systemfunktion, die den Typ eines Ausdrucks bestimmt. Sie kann benutzt werden, um die G.ltigkeit eines Ausdrucks zu testen. Dieser kann Funktionen aus der Bibliothek CLIPPER.LIB enthalten. Er darf jedoch nicht auf lokale oder statische Variablen verweisen und keine User Defined Functions oder Funktionen aus der Bibliothek EXTEND.LIB enthalten.

TYPE() ist .hnlich der Funktion VALTYPE(), differiert aber durch die Verwendung eines Makro-Operators (&), um den Typ eines Ausdrucks zu ermitteln. Diese Eigenschaft hindert TYPE() daran, den Typ von lokalen und statischen Variablen zu bestimmen. Im Gegensatz dazu ermittelt VALTYPE() den Datentyp dadurch, da. der Ausdruck berechnet und der Typ des R.ckgabewertes gepr.ft wird. Dies erlaubt auch die .berpr.fung von User Defined Functions und von lokalen und statischen Variablen.

Hinweise

. Referenzen auf Arrays: Referenzen auf Arrays werden durch "A" gekennzeichnet. Referenzen auf ein Element eines Arrays, geben den Typ des Elementes zur.ck.

. IF(): Um auch in Zusammenhang mit der Funktion IF() korrekt zu arbeiten, wertet TYPE() die Bedingung zun.chst aus und bestimmt dann den Datentyp des resultierenden Ausdrucks. Ist IF() oder das Ergebnis fehlerhaft, wird "UE" zur.ckgegeben.

. Testen von Parametern: TYPE() kann lediglich die G.ltigkeit von Parametern bestimmen, wenn diese .ber die Anweisung PARAMETERS entgegengenommen wurden.
Parameter, die in einer FUNCTION- oder PROCEDURE-Deklaration entgegengenommen wurden, ergeben den Typ "U". Um zu testen, ob Parameter .bersprungen oder am Ende der Parameterliste weggelassen wurden, kann der Parameter mit NIL verglichen oder mit VALTYPE() gepr.ft werden (siehe auch die Beispiele weiter unten).

. User Defined Functions und Funktionen aus EXTEND.LIB: Wird in <cAusdruck> auf eine Funktion verwiesen, die nicht in CLIPPER.LIB enthalten ist (anwenderdefinierte Funktionen oder Funktionen aus EXTEND.LIB), wird "UI" zur.ckgegeben. Ist die User Defined Function nicht zusammen mit Ihrer Applikation gelinkt worden, liefert TYPE() den

R.ckgabewert "U".

Beispiele

. Das folgende Beispiel zeigt die verschiedenen Ergebnisse durch den Aufruf der TYPE()-Funktion:

```
? TYPE('SUBSTR("Hallo Ihr da", 4, 5)')  // Resultat: C
? TYPE("UDF()")                         // Resultat: UI
? TYPE('IF (.T., "wahr", "falsch")')    // Resultat: C
```

. Das Beispiel zeigt zwei Methoden, um die Existenz und den Typ von Parametern zu pr.fen:

```
FUNCTION ParaTest( cParam, nParam2 )
  IF cParam1 = NIL
    ? "kein erster Parameter .bergeben !"
    cParam1 := "Vorgabewert"
  ENDIF
  //
  IF VALTYPE( nParam2 ) <> "N"
    ? "ung.ltiger oder falscher Parameter !"
    nParam2 := nVorgabe
  ENDIF
  .
  .<Anweisungen>...
  .
RETURN NIL
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

UPDATED() Feststellen, ob ein GET-Objekt w.hrend eines READ's ge.ndert wurde

---

Syntax

UPDATED() -> lGe.ndert

R.ckgabe

UPDATED() liefert true (.T.), wenn ein GET-Objekt ge.ndert wurde, und false (.F.), wenn nicht.

Anwendung

UPDATED() wird ben.tigt, um festzustellen, ob durch Eingaben in GET-Objekte eines Eingabeformulars w.hrend eines READs der urspr.ngliche Inhalt ge.ndert wurde. Jedesmal, wenn ein READ ausgef.hrt wird, ist der Wert von UPDATED() false (.F.). Sobald dann eine Eingabe in eines der aktuellen GET-Objekte gemacht wird, wechselt der Status von UPDATED() auf true (.T.). Verl..t der Benutzer das erste Eingabefeld durch Bet.tigen von [Esc], wird UPDATED() auf false (.F.) zur.ckgesetzt. Der Status von UPDATE() bleibt so lange erhalten, bis das n.chste READ ausgef.hrt wird.

GET-Objekte k.nnen auch in Prozeduren oder Funktionen ge.ndert werden, die .ber SET KEY oder VALID aufgerufen werden. Diese .nderung kann wiederum .ber den Befehl KEYBOARD oder durch die Zuweisung eines neuen Wertes .ber einen Zuweisungsoperator erfolgen. .nderungen .ber KEYBOARD sind genauso, als h.tte der Benutzer die Werte .ber die Tastatur eingegeben und UPDATE() wird korrekt gesetzt. Die Neuzuweisung eines Wertes, die nicht .ber KEYBOARD erfolgt, hat keinen Einflu. auf den Status in UPDATE().

Beispiele

Im folgenden Beispiel werden einige Speichervariablen initialisiert und editiert. Der Datensatz wird nur dann aktualisiert, wenn der Benutzer eine Eingabe vornimmt.

```
USE Kunden NEW
CLEAR
MEMVAR->Name := Kunden->Name
MEMVAR->Adresse := Kunden->Adresse
MEMVAR->Telefon := Kunden->Telefon
@ 1, 1 SAY "Name:     " GET MEMVAR->Name
@ 2, 1 SAY "Adresse:  " GET MEMVAR->Adresse
READ
//
IF UPDATED()
                              Kunden->Name := MEMVAR->Name
                              Kunden->Adresse := MEMVAR->Adresse
ENDIF
```

Dateien **Bibliothek ist CLIPPER.LIB**

UPPER() Wandelt Klein- in Gro.buchstaben um

Syntax

UPPER(<cZeichenkette>) -> cZeichenkette

Parameter

<cZeichenkette> ist die umzuwandelnde Zeichenkette.

R.ckgabe

UPPER() gibt eine Zeichenkette zur.ck, die anstelle von Kleinbuchstaben Gro.buchstaben enth.lt.

Anwendung

UPPER() ist eine Zeichenfunktion, die alle, in einer Zeichenkette enthaltenen Kleinbuchstaben in Gro.buchstaben umwandelt. Sie ist vergleichbar mit LOWER(), wo alle Gro.buchstaben in Kleinbuchstaben umgewandelt werden. Die Funktion ist ebenfalls vergleichbar mit den Funktionen ISUPPER() und ISLOWER(), bei denen getestet wird, ob eine Zeichenkette mit einem Gro.- oder Kleinbuchstaben beginnt.

UPPER() wird normalerweise benutzt, um Zeichenketten formatiert auszugeben. Die Funktion kann auch in Indexausdr.cken verwendet werden, um von der Gro.-/Kleinschreibung unabh.ngige Indexschl.ssel zu bilden.

Beispiele

. Das folgende Beispiel verdeutlicht den Effekt der Funktion UPPER():

```
? UPPER("eine Zeichenkette")  // Resultat: EINE ZEICHENKETTE
? UPPER("123 Zeichen = <>")   // Resultat: 123 ZEICHEN = <>
```

. Hier wird gezeigt, wie die Gro.-/Kleinschreibung unabh.ngiger Indexschl.ssel mit UPPER() erstellt werden kann:

```
USE Kunden NEW
INDEX ON UPPER(Name) TO NameGros
```

. Sp.ter kann .ber den gleichen Ausdruck ein Begriff gesucht werden:

```
LOCAL cSuch := SPACE(15)
@ 10, 10 SAY "Suchbegriff: " GET cSuch
READ
SEEK UPPER(cSuch)
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

USED() Ermittelt, ob eine Datenbank ge.ffnet ist

Syntax

USED() -> lGe.ffnet

R.ckgabe

USED() liefert true (.T.), wenn eine Datenbank ge.ffnet ist, ansonsten wird false (.F.) zur.ckgegeben.

Anwendung

USED() ist eine Datenbankfunktion und wird benutzt, um festzustellen, ob in einem Arbeitsbereich eine Datenbank offen ist. Als Voreinstellung pr.ft USED() den aktuellen Arbeitsbereich. Um diese Funktion f.r andere Arbeitsbereiche anzuwenden, mu. ein Aliasbezeichner vorangestellt werden.

Beispiele

. In diesem Beispiel wird ermittelt, ob in dem aktuellen Arbeitsbereich eine Datenbank ge.ffnet worden ist:

```
USE Kunden NEW
? USED()  // Resultat: .T.
CLOSE
? USED()  // Resultat: .F.
```

. Im n.chsten Beispiel wird gepr.ft, ob eine bestimmte Datenbank ge.ffnet worden ist:

```
USE Kunden NEW
USE Verkauf NEW
USE Rechnung NEW
SELECT 0
//
? USED()               // Resultat: .F.
? Kunden->(USED())     // Resultat: .T.
? verkauf->(USED())    // Resultat: .T.
? rechnung->(USED())   // Resultat: .T.
? auftrag->(USED())    // Resultat: .F.
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

VAL() Konvertiert eine Zeichenkette in einen numerischen Datentyp

---

Syntax

VAL(<cZahl>) -> nZahl

Parameter

<cZahl> ist der zu konvertierende Zeichenkettenausdruck.

R.ckgabe

VAL() liefert den Wert in <cZahl> als numerischen Wert inclusive Dezimalstellen.

Anwendung

VAL() ist eine Zeichen-Konvertierungsfunktion. Sie wandelt einen Zeichenkettenausdruck in einen .quivalenten numerischen Wert um. Die Umwandlung beginnt links in der Zeichenkette. Der in der Zeichenkette befindliche numerische Ausdruck gilt als terminiert, wenn ein zweiter Dezimalpunkt oder ein nicht als Zahl identifizierbares Zeichen auftritt, oder wenn das Ende der Zeichenkette erreicht ist. F.hrende Leerzeichen werden nicht beachtet. Ist SET FIXED auf ON gestellt, gibt VAL() so viele Dezimalstellen zur.ck, wie in SET DECIMALS angegeben worden sind. Sind weniger Dezimalstellen angegeben, als das Ergebnis von <cZahl> ergibt, wird der R.ckgabewert gerundet. Wie bei allen anderen Funktionen, die einen Wert runden, werden Werte kleiner f.nf abgerundet und Werte gr..er oder gleich f.nf aufgerundet.

VAL() ist die gegens.tzliche Funktion zu STR(), die numerische Werte in Zeichenketten wandelt.

Beipiele:

Die folgenden Beispiele zeigen das Ergebnis von VAL() bei SET FIXED ON und SET DECIMALS TO 2:

```
SET DECIMALS TO 2
SET FIXED ON
? VAL("12.1234")  // Resultat: 12.12
? VAL("12.1256")  // Resultat: 12.13
? VAL("12A12")    // Resultat: 12.00
? VAL("A1212")    // Resultat: 0.00
? VAL(SPACE(1))   // Resultat: 0.00
? VAL(" 12.12")   // Resultat: 12.12
? VAL("12 .12")   // Resultat: 12.00
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

VALTYPE() Ermittelt den Datentyp eines berechneten Ausrucks

Syntax

VALTYPE(<Ausdruck>) -> cTyp

Parameter

<Ausdruck> ist ein Ausdruck beliebigen Typs.

R.ckgabe

VALTYPE() liefert ein einzelnes Zeichen, das den Datentyp des berechneten Ausdrucks in <Ausdruck> angibt. Die folgende Liste enth.lt die m.glichen R.ckgabewerte:

Tabelle 5-37: VALTYPE() R.ckgabewerte

R.ckgabewert	Bedeutung
A	Array
B	Block
C	Zeichen
D	Datum
L	Logisch
M	Memo
N	Numerisch
U	NIL

Anwendung

VALTYPE() ist eine Clipper-Systemfunktion, die einen .bergebenen Ausdruck berechnet, den Typ des Ergebniswertes ermittelt und als Funktionsergebnis ein einzelnes Zeichen liefert, welches den Typ beschreibt. Sie ist .hnlich der Funktion TYPE(), berechnet aber zuerst den Ausdruck, bevor der Typ ermittelt wird. Auf diese Weise k.nnen auch Datentypen von lokalen und statischen Variaben, von User Defined Functions und von Funktionen, die nicht in CLIPPER.LIB enthalten sind, ermittelt werden. TYPE() dagegen benutzt den Makro-Operator (&) zur Ermittlung des Typs.

Beispiele

Die folgenden Beispiele zeigen den R.ckgabewert verschiedener Datentypen:

```
? VALTYPE("Hallo")  // Resultat: C
? VALTYPE(NIL)      // Resultat: U
? VALTYOE(block)    // Resultat: B
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

VERSION() Ermittelt die Clipper Version

Syntax

VERSION() -> cClipperVersion

R.ckgabe

VERSION() ermittelt die Versionsnummer der Clipper-Bibliothek EXTEND.LIB als Zeichenwert.

Anwendung

VERSION() ist eine Enviroment-Funktion, welche die Versionsnummer der Clipper-Bibliothek EXTEND.LIB zur.ckgibt.

Dateien

WORD()* Umwandlung numerischer Parameter f.r CALL-Befehle von double nach int

---

Syntax

WORD(<nZahl>) -> NIL

Parameter

<nZahl> ist der numerische Wert, der nach int konvertiert werden soll. Er kann im Bereich zwischen -32.768 und + 32.767 liegen.

R.ckgabe

Als Argument in einem CALL-Befehl liefert WORD() den Wert von <nZahl> als integer zur.ck, in allen anderen F.llen wird NIL zur.ckgegeben.

Anwendung

WORD() ist eine numerische Konvertierungsfunktion, anwendbar als numerischer Parameter im CALL-Befehl, um interne Zahlenformatdarstellungen vom Typ double nach integer zu wandeln. WORD() ist eine Funktion, die aus Kompatibilit.tsgr.nden implementiert ist. Die Anwendung wird daher nicht empfohlen. Beides, die WORD()-Funktion und der CALL-Befehl k.nnen durch M.glichkeiten des Extend-Systems ersetzt werden. Lesen Sie dazu bitte auch im Handbuch Programmierung und Hilfsprogramme im Kapitel Das Extend System nach, um n.here Informationen zu erhalten.

Beispiele

Das Beispiel zeigt eine typische Anwendung von WORD() als Parameter eines CALL-Befehls:

```
CALL Cprozedur WITH WORD(3210), "irgend ein Text"
```

Dateien

YEAR() Konvertiert einen Datumswert in eine numerische Jahreszahl

---

Syntax

YEAR(<dDatum>) -> nJahr

Parameter

<dDatum> ist der zu konvertierende Datumswert.

R.ckgabe

YEAR() liefert die Jahreszahl (inkl. Jahrhundert) eines angegebenen Datums als vierstelligen, numerischen Wert. Der R.ckgabewert wird nicht von SET DATE oder CENTURY beeinflu.t. Die Angabe eines leeren Datums (CTOD("")) ergibt eine Null.

Anwendung

YEAR() ist eine Datum-Konvertierungsfunktion und wird benutzt, um die Jahreszahl eines Datums zu ermitteln. Sie kann innerhalb von Berechnungen von Nutzen sein, wenn zum Beispiel periodische Auswertungen erstellt werden sollen. YEAR() bildet mit MONTH() und SAY() eine Gruppe von Funktionen, die Teile eines Datums in einen numerischen Wert wandeln.

Beispiele

. Das folgende Beispiel beschreibt YEAR() angewendet auf das Systemdatum:

```
? DATE()                // Resultat: 01.09.90
? YEAR(DATE())          // Resultat: 1990
? YEAR(date()) + 11     // Resultat: 2001
```

. Das n.chste Beispiel zeigt eine User Defined Function, die das Datum in die Form "Tag. Monat Jahr" bringt:

```
FUNCTION MyDate( dDatum )
RETURN STR(DAY(dDatum),2) + ". " + CMOMTH(dDatum) + " " + ;
STR(YEAR(dDatum),4)
```

. Sp.ter kann die Funktion wie folgt angewendet werden:

```
LOCAL cDatum := MyDate( DATE())
? cDatum  // Resultat: 1. September 1990
//
@ 1, 80-LEN(cDatum) SAY cDatum
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

@...BOX Zeichnet eine Box auf dem Bildschirm

Syntax

@ <nOben>, <nLinks>, <nUnten>, <nRechts> BOX <cBoxString>

Argumente

<nOben>, <nLinks>, <nUnten>, <nRechts> geben die Koordinaten der Box an. Mit @...Box k.nnen Boxen mit Spaltenwerten von 0 bis MAXCOL() und Zeilenwerten von 0 bis MAXROW() gezeichnet werden. Wenn <nUnten> und <nRechts> gr..er als MAXROW() und MAXCOL() angegeben werden, verschwindet die rechte untere Ecke aus dem Bildschirm.

<cBoxString> ist eine Zeichenkette mit acht Rahmen- und einem F.llzeichen. Wird dieses Argument mit nur einem Zeichen angegeben, so wird dieses f.r die gesamte Box verwendet.

Anwendung

@...BOX zeichnet unter Verwendung individueller Rahmen- und F.llzeichen eine Box auf dem Bildschirm. Dabei wird die Box aus dem angegebenen <cBoxstring> von der linken oberen Ecke aus im Uhrzeigersinn aufgebaut und das Box-Innere dann mit dem F.llzeichen gef.llt. Wird das neunte Zeichen (F.llzeichen) nicht angegeben, so werden innerhalb der Box keine Ausgaben durchgef.hrt. Text und Farben dieses Bildschirmbereiches bleiben dann erhalten.
Nach dem Aufbau der Box steht der Cursor in der linken oberen Ecke der Box an den Koordinaten <nOben>+1 und <nLinks>+1. ROW() und COL() werden entsprechend aktualisiert.

Beispiele

Im folgenden Beispiel werden zwei Boxen unter Benutzung vordefinierter Konstanten aus der Header-Datei Box.CH angezeigt. Im ersten Beispiel wird nur die Box gezeichnet, der Rest des Bildschirms bleibt unver.ndert. Im zweiten Beispiel hingegen wird der Bereich innerhalb der Box mit Leerzeichen gef.llt.

```
#include "Box.CH"

// Box mit waagerechten Doppellinien und senkrechten Einfachlinien
@ 1, 1, 11, 79 BOX B_DOUBLE_SINGLE
// Die gleiche Box, jedoch mit Leerzeichen gef.llt
@ 1, 1, 11, 79 BOX B_DOUBLE_SINGLE + SPACE(1)
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist Box.CH

See Also:

@...CLEAR L.scht einen rechteckigen Bereich des Bildschirms

Syntax

@ <nOben>, <nLinks> [CLEAR [TO <nUnten>, <nRechts>]]

Argumente

<nOben>, <nLinks> geben die Koordinate der oberen linken Ecke an.

<nUnten>, <nRechts> geben die Koordinate der rechten unteren Ecke f.r den zu l.schenden Bildschirmbereich an. Wurde TO nicht angegeben, so wird bis MAXROW() und MAXCOL() gel.scht.

Anwendung

@...CLEAR l.scht einen rechteckigen Bereich innerhalb des Bildschirms, indem dieser mit Leerzeichen unter Ber.cksichtigung des eingestellten Farbattributs gef.llt wird. Nach dem L.schen des Bereiches steht der Cursor in der linken oberen Ecke desselben auf den Koordinaten <nOben>+1 und <nLinks>+1. ROW() und COL() werden entsprechend aktualisiert. Wenn Sie als Argumente nach @ nur <nOben> und <nLinks> angeben, so wird die Zeile <nOben> ab der Spalte <nLinks> bis zum Zeilenende MAXCOL() gel.scht.

Beispiele

. Das folgende Beispiel l.scht den Bildschirm im Bereich 10, 10 bis 20, 40, f.llt ihn mit der Farbe blau und zeichnet dann eine Box in Cyan auf Blau:

```
SETCOLOR("BG+/B")
@ 10, 10 CLEAR TO 20, 40
@ 10, 10 TO 20, 40
```

. Dieses Beispiel l.scht Zeile 4 ab Spalte 10 bis zum Zeilenende:

```
@ 04, 10
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

@...GET Legt ein neues GET-Objekt an und gibt es auf dem Bildschirm aus

---

Syntax

```
@ <nZeile>, <nSpalte>
  [SAY <Ausdruck>
    PICTURE <cSayPicture>]]
  GET  <idVar>
    [PICTURE <cGetPicture>]
    [WHEN <lPreBedingung>]
    [RANGE <dnUntergrenze>, <dnObergrenze>] | [VALID <lPostBedingung>]
```

Argumente

<nZeile> und <nSpalte> geben die Anfangszeile und -spalte f.r die Operation an. Wurde SAY angegeben, so ist dies die erste Position f.r die damit verbundene Ausgabe. Das GET-Eingabefeld wird, getrennt durch das Begrenzerzeichen, dahinter angezeigt. Ohne die Angabe von SAY spezifizieren die beiden Werte direkt die Position des GET-Feldes. In beiden F.llen werden Ausgaben au.erhalb des sichtbaren Bildschirmbereiches abgeschnitten.

SAY <Ausdruck> zeigt den Wert von <Ausdruck> auf den vorgegebenen Koordinaten an. Wurde PICTURE <cSayPicture> angegeben, so wird die Ausgabe entsprechend formatiert.

GET <idVar> benennt den mit dem Eingabefeld verbundenen Variablennamen. <idVar> darf jeder beliebigen Speicherklasse (LOCAL etc.) angeh.ren. Ist diese jedoch unklar, so wird ein Datenbankfeld (FIELD) angenommen. Wenn <idVar> ein Arraywert ist, m.ssen ein oder mehrere Array-Indizes angegeben werden. Der aktuelle Wert von <idVar> wird an den GET-Koordinaten angezeigt. Dieser Wert mu. ein Zeichen-, Datums-, numerischer oder logischer Typ sein. Arrays, NIL oder Code-Bl.cke sind nicht erlaubt.

PICTURE <cGetPicture> spezifiziert die Anzeige und Eingaberegeln f.r das GET-Feld.

WHEN <lPreBedingung> gibt einen logischen Ausdruck an, der true (.T.) liefern mu., damit das zugeh.rige GET-Feld aktiviert wird.

VALID <lPostBedingung> gibt einen logischen Ausdruck an, der true (.T.) liefern mu., damit das zugeh.rige GET-Feld verlassen werden kann.

RANGE <dnUntergrenze>, <dnObergrenze> gibt einen G.ltigkeitsbereich f.r die Eingaben des zugeh.rigen GET-Feldes an. Dieser Bereich darf durch numerische oder Datumswerte eingegrenzt sein. W.hrend eines READ kann das Eingabefeld so lange nicht verlassen werden, wie keine Eingabe innerhalb des zul.ssigen Bereiches erfolgte. VALID und RANGE k.nnen nicht gemeinsam verwendet werden.

Anwendung

Der @...GET-Befehl erzeugt ein neues GET-Objekt, zeigt dessen zugeordnete Daten an und f.gt es dem Array GetList hinzu. Wenn in der aktiven Prozedur keine Variable mit Namen GetList deklariert oder erzeugt worden ist und keine PRIVATE-Variable mit diesem Namen von einer aufrufenden Prozedur her existiert, wird die vordefinierte PUBLIC-Variable GetList verwendet. Ein nachfolgender READ-Befehl aktiviert die GET-Felder innerhalb des Arrays GetList und gestattet es dem Anwender, Eingaben vorzunehmen.

Zu jedem GET-Objekt existiert eine Eingabevariable, die hier mit <idVar> angegeben wird. Diese Variable darf jeder der Speicherklassen PRIVATE, PUBLIC, LOCAL, STATIC oder FIELD angeh.ren. Wird die Variable <idVar> indiziert, so wird das entsprechende Array-Element mit dem GET-Objekt verbunden. Bei Erzeugen des GET-Objektes wird der Name von <idVar> innerhalb des Objektes gespeichert und mit einem Code-Block verbunden, um ihn w.hrend eines READ jederzeit abfragen oder zuordnen zu k.nnen.

Der READ-Befehl f.hrt alle GET-Felder innerhalb des GetList-Arrays als

gemeinsame Full-Screen-Operation aus. Wird durch die Cursorbewegungen des Anwenders das GET-Feld gewechselt, so l..t sich mit Hilfe des Code-Blocks, der innerhalb des GET-Objekts gespeichert ist, der Inhalt der jeweils zugeh.rigen Variablen abfragen. Dieser Wert wird in einen String umgeformt und dann in einem Puffer innerhalb des GET-Objekts abgelegt. Genau dieser Pufferinhalt wird auf dem Bildschirm angezeigt und kann .ber Tastatureingaben bearbeitet werden. Bei Verlassen des Eingabefeldes wird der ver.nderte Inhalt dann wieder in den entsprechenden Datentyp zur.ckverwandelt und in <idVar> abgelegt.

Automatische Formatierung und G.ltigkeitspr.fung: Einige Formatierungen und G.ltigkeitspr.fungen f.r die Datentypen numerisch, logisch und Datum werden w.hrend eines READ automatisch durchgef.hrt. Jedes eingegebene Zeichen wird auf seine Zul.ssigkeit .berpr.ft, so da. der Anwender keine falschen Zeichen eingeben kann.

Pre-G.ltigkeitspr.fung: Die WHEN-Klausel spezifiziert eine logische Bedingung, die erf.llt sein mu., um in das zugeh.rige GET-Feld zu gelangen. W.hrend eines READ wird <lPreBedingung> jedesmal .berpr.ft, wenn der Anwender den Cursor in dieses Feld bewegen will. Liefert der Ausdruck true (.T.), so wird das Feld aktiv, bei false (.F.) hingegen wird es .bersprungen.

Post-G.ltigkeitspr.fung: Diese Art der .berpr.fung kann entweder durch VALID oder durch RANGE erfolgen. In Verbindung mit VALID ist ein logischer Ausdruck anzugeben, der erf.llt sein mu., damit das Eingabefeld verlassen werden kann. W.hrend eines READ wird <lPostBedingung> jedesmal .berpr.ft, wenn der Anwender den Cursor aus diesem Feld herausbewegen will. Liefert der Ausdruck true (.T.), darf das Feld verlassen werden, anderenfalls nicht. RANGE hingegen gibt einen Bereich zul.ssiger Werte f.r numerische und Datumseingaben an. Liegt die Eingabe des Anwenders nicht innerhalb dieses Bereiches, so kann das Feld nicht verlassen werden.

PICTURE: Wird f.r ein GET-Feld die PICTURE-Klausel angegeben, so spezifiziert der String <cGetPicture> nicht nur die Formatierung des Feldes, sondern auch die erlaubten Eingabezeichen. Die Kontrolle der Formatierung erfolgt dabei .hnlich der PICTURE-Klausel von SAY, zus.tzlich werden aber noch alle Anwender-eingaben kontrolliert. Ein PICTURE-String besteht aus zwei wesentlichen Teilen, die einzeln oder zusammen angegeben werden k.nnen: der Formatierungsfunktion und der Formatierungsschablone.

. **Formatierungsfunktion:** Eine PICTURE-Formatierungsfunktion spezifiziert eine bestimmte Formatierung oder G.ltigkeitspr.fung f.r das ganze Feld, anstatt f.r ein einzelnes Zeichen innerhalb des Feldes. Eine solche Formatierungsfunktion beginnt immer mit dem Zeichen @, gefolgt von einem oder mehreren Buchstaben, von denen wiederum jeder eine besondere Bedeutung hat (s. folgende Tabelle). Wird eine Formatierungsfunktion angegeben, so mu. das @-Zeichen immer das erste innerhalb des PICTURE-Strings sein, und innerhalb der Formatierungsfunktion d.rfen keine Leerzeichen enthalten sein. Eine Formatierungsfunktion kann allein, aber auch gemeinsam mit einer PICTURE-Schablone angegeben werden. Werden beide Arten verwendet, so mu. die Formatierungsfunktion zuerst angegeben sein, und beide m.ssen durch ein Leerzeichen voneinander getrennt werden.

Tabelle 4-1: GET PICTURE Formatierungsfunktionen

Funkt.	Typ	Auswirkung
A	C	Nur Buchstaben erlaubt
B	N	Zahlen werden nach links ausgerichtet
C	N	Nach positiven Zahlen CR (Credit) anzeigen
D	D,N	Datum im SET DATE-Format anzeigen
E	D,N	Datum und Zahlen in (E)urop.ischer Schreibweise anzeigen
K	ALLE	L.scht Vorgabetext, wenn erste Taste keine Cursortaste ist
R	C	Zeichen, die keine Schablonen-Zeichen sind, werden angezeigt, jedoch nicht in die Variable .bernommen und gespeichert
S.C	Ein Feld, gr..er als der zur Anzeige genutzte Bereich, wird	

		horizontal gescrollt. gibt dabei die Gr..e des sichtbaren Bereiches an.
X	N	Nach negativen Zahlen DB (Debit) anzeigen
Z	N	Anstelle f.hrender Nullen Leerzeichen anzeigen
(	N	Negative Zahlen werden einschlie.lich f.hrender Leerstellen in runden Klammern angezeigt
)	N	Negative Zahlen werden ohne f.hrende Leerstellen in runden Klammern angezeigt
!	C	Alle Kleinbuchstaben werden in Gro.buchstaben umgewandelt (UPPER CASE)

. **Formatierungsschablone:** Eine PICTURE-Formatierungsschablone gibt die Formatierungen und Regeln zur G.ltigkeitspr.fung f.r jedes einzelne Zeichen des Eingabefeldes an. Dabei sind eine Reihe von Schablonenzeichen mit besonderer Bedeutung definiert (s. folgende Tabelle). Jede Position in der Formatierungsschablone korrespondiert genau mit einer Stelle im aktiven Eingabefeld. Zeichen, die in der Formatierungsschablone keine besondere Bedeutung haben, werden ohne Ver.nderung oder Interpretation angezeigt. Wenn gleichzeitig die PICTURE-Funktion @R verwendet wird, so werden diese Zeichen zwischen denen der Eingabevariablen eingef.gt, angezeigt und automatisch wieder gel.scht, wenn die ver.nderte Eingabe wieder <idVar> zugewiesen wird. Anderenfalls .berschreiben solche Zeichen die korrespondierenden Daten und beeinflussen so letztendlich auch den Inhalt von <idVar>. Eine Formatierungsschablone kann allein, aber auch gemeinsam mit einer PICTURE-Funktion angegeben werden. Werden beide Arten verwendet, so mu. die Formatierungsfunktion zuerst angegeben sein, und beide m.ssen durch ein Leerzeichen voneinander getrennt werden.

Tabelle 4-2: GET PICTURE Schablonen-Symbole

Schablonen	Auswirkung
A	Nur Buchstaben sind erlaubt
N	Nur Buchstaben und Ziffern sind erlaubt
X	Beliebige Zeichen sind erlaubt
9	Erlaubt Ziffern f.r jeden Datentyp, f.r numerische Eingaben auch das Minus-Zeichen
#	Erlaubt Ziffer, Minus-Zeichen und Leerzeichen f.r jeden Datentyp
L	Erlaubt nur T, F, J oder N
Y	Erlaubt nur J oder N
!	Wandelt Klein- in Gro.buchstaben um
$	Zeigt bei numerischen Datentypen anstelle eines f.hrenden Leerzeichens ein $-Zeichen an
*	Zeigt bei numerischen Datentypen anstelle eines f.hrenden Leerzeichens das Zeichen * an
.	Zeigt einen Dezimalpunkt an
,	Zeigt ein Komma an

SCOREBOARD: Mu. eine Neueingabe zur.ckgewiesen werden, weil sie au.erhalb des zul.ssigen RANGE-Bereiches liegt oder weil es sich um ein ung.ltiges Datum handelt, so erscheint eine Fehlermeldung auf dem Bildschirm. Dies geschieht innerhalb des SCOREBOARD-Bereiches (oberste Bildschirmzeile) und kann .ber den Befehl SET SCOREBOARD OFF abgeschaltet werden.

Verlassen .ber [Esc]: Wenn ein Eingabefeld durch [Esc]verlassen wird, so wird die Eingabevariable auf ihren urspr.nglichen Wert zur.ckgesetzt und das READ beendet. Eine Post-G.ltigkeitspr.fung wird in diesem Fall nicht durchgef.hrt. Dieses Verhalten kann .ber den SET ESCAPE-Befehl ein- und ausgeschaltet werden.

SET KEY-Prozeduren: Der Befehl SET KEY erlaubt es, eine Prozedur anzugeben, die w.hrend eines READ immer dann aufgerufen wird, wenn eine bestimmte Taste bet.tigt wurde. Sobald die SET KEY-Prozedur beendet ist, wird das GET-Feld wieder aktiv und die vorherige Cursor-Position wiederhergestellt.

Existenzdauer eines GET-Objektes: Get-Objekte existieren genau wie Arrays so lange, wie irgendwo im Programm auf sie Bezug genommen wird.

Normalerweise h.ngen GET-Objekte nur von der Existenz des Arrays GetList ab; wird das Array verworfen oder neu zugeordnet, gehen auch alle GET-Objekte verloren. Die Befehle CLEAR und CLEAR GETS legen jeweils ein neues, leeres Array GetList an. Dies gilt ebenfalls f.r den READ-Befehl, es sei denn, der Zusatz SAVE wurde verwendet.

Zuweisungen: Jedes GET ist mit einer Variablen verbunden, die beim @...GET-Befehl f.r <idVar> anzugeben ist. W.hrend des Bearbeitungsprozesses wird <idVar> an verschiedenen Stellen der aktuelle Inhalt des GET-Objektpuffers zugewiesen. Dies geschieht in folgenden F.llen:

. Nachdem der Anwender eine Taste bet.tigt, mit der das Eingabefeld verlassen wird, jedoch vor der Post-G.ltigkeitspr.fung.

. Wenn der Anwender eine .ber SET KEY definierte Taste bet.tigt.

Entsprechend existieren auch Situationen, in denen der Puffer des GET-Objektes mit dem Inhalt von <idVar> wieder aufgefrischt und in verschiedenen Intervallen wieder angezeigt wird:

. Nachdem eine SET KEY-Prozedur beendet wurde.

. Nachdem die WHEN-Bedingung ausgewertet wurde.

. Nachdem die VALID-Bedingung ausgewertet wurde.

Dies erlaubt es Ihnen, die Variable beispielsweise in einer SET KEY-Prozedur zu ver.ndern bzw. ihr einen neuen Wert zuzuordnen. GET-Koordinaten und -Anzeige: Wird ein neues GET-Objekt .ber den @...GET-Befehl angelegt, so werden auch die Zeilen- und Spaltenkoordinaten, an denen das Eingabefeld angezeigt wird, innerhalb des GET-Objektes abgelegt. Wenn der @...GET-Befehl ausgef.hrt wird, so erfolgt die Anzeige bei <nZeile> und <nSpalte>, wenn kein SAY angegeben wurde. Anderenfalls wird das Eingabefeld (oder der linke DELIMITER, s.u.) hinter die durch SAY erfolgte Ausgabe, allerdings getrennt durch ein zus.tzliches Leerzeichen, positioniert.

Wurde SET DELIMITERS durch ON eingeschaltet, so werden die jeweils definierten Begrenzungszeichen links und rechts des GET-Feldes dargestellt, wobei die Anfangsposition des Feldes entsprechend korrigiert wird. Beachten Sie bitte, da. die DELIMITER nicht Teil der GET-Objekte sind, sondern einfache Ausgabezeichen (wie durch ein SAY) darstellen.

Wurde SET INTENSITY durch ON eingeschaltet, dann werden die Felder zuerst in der SET COLOR-Farbspezifikation unselected dargestellt (falls f.r unselected keine Einstellung erfolgt ist, so wird die Farbe enhanced verwendet). W.hrend eines READ wird das jeweils aktive Feld dann in der Farbe enhanced gezeigt, w.hrend alle anderen Felder als unselected dargestellt werden. Wird SET INTENSITY ausgeschaltet, erscheinen alle Felder in der Farbe standard.

Bei der Anzeige eines GET-Feldes wird dessen Ausgabel.nge entweder durch den Wert der Eingabevariablen <idVar> oder aber durch die Anzahl der Positionen in einer gegebenenfalls verwendeten PICTURE-Klausel <cGetPicture> bestimmt. Wurde allerdings die @S-Funktion verwendet, richtet sich die L.nge des Feldes ausschlie.lich nach dem dort angegebenen Wert.

Hinweise

. **WHEN und VALID:** Die Ausdr.cke, die in Verbindung mit WHEN und VALID angegeben werden k.nnen, d.rfen von beliebiger Komplexit.t sein und demnach auch User Defined Functions (UDFs) enthalten. Dies ist sehr n.tzlich, um so beliebige Aktionen mit der Aktivierung und Deaktivierung eines GET-Feldes zu verbinden.

. **Zuweisungen von <idVar>:** Durch die Aktualisierungs-Automatik und das

Anzeigeverhalten eines GET-Objektes w.hrend eines READ kann die Eingabevariable des Feldes auch innerhalb des WHEN- oder VALID-Ausdrucks ver.ndert werden. Die Variable kann innerhalb des Ausdrucks direkt .ber ihren Namen ver.ndert werden. Bei LOCAL-, STATIC-, PUBLIC- und PRIVATE-Variablen kann dies auch durch die .bergabe einer Referenz auf <idVar> an eine Funktion geschehen. Innerhalb der Funktion wird <idVar> dann durch die Zuweisung an den entsprechenden formalen Parameter beeinflu.t. Wenn es sich bei <idVar> um ein Datenbankfeld handelt, ist dieses global verf.gbar und kann immer direkt .ber seinen Namen ver.ndert werden.

. **Individuelle Hilfe f.r ein GET-Feld:** Hilfetexte, die mit einem GET-Feld in Verbindung stehen, k.nnen durch die Verwendung einer SET KEY-Prozedur angezeigt werden. Innerhalb dieser SET KEY-Prozedur kann .ber die Funktion READVAR() der Name der aktuellen Eingabevariablen ermittelt und so der zugeh.rige Hilfetext bespielsweise aus einer Datenbank abgerufen werden. Wurde bei einem Clipper-Programm nichts anderes angegeben, ist die Taste [F1] automatisch mit einer Prozedur HELP verkn.pft.

. **SET DEVICE TO PRINTER:** Die Ausgabe eines GET-Feldes wird nicht auf den Drucker umgelenkt, auch dann nicht, wenn SET DEVICE TO PRINTER eingestellt wurde.

See Also:

Beispiele

. Das folgende Beispiel demonstriert die Anwendung der VALID-Klausel f.r die .berpr.fung der Eingabe bei Verlassen des Feldes:

```
LOCAL nZahl := 0
@ 10, 10 SAY "Bitte eine Zahl eingeben:" GET nZahl;
  VALID nZahl > 0
```

. Im folgende Beispiel wird die Eingabe einer neuen Kundennummer .berpr.ft. Ist diese bereits vorhanden, wird eine Meldung ausgegeben und der Benutzer ist gezwungen eine korrekte Eingabe vorzunehmen:

```
@ 10, 10 SAY "Neue Kundennummer: " GET KundenNr VALID Kd_Ok(KundenNr)
READ
FUNCTION Kd_Ok( KdNr )
  LOCAL nOldRec := RECNO(), lGueltig
  SEEK KdNr
  IF !(lGueltig := !FOUND())
    @ 23,0 SAY "Kundennummer bereits vorhanden!"
    INKEY(5)
    @ 23, 0
  ENDIF
  GOTO nOldRec
  RETURN lGueltig
```

. Dieses Beispiel zeigt, wie durch die WHEN-Klausel der Eintritt in ein GET-Feld, abh.ngig von den Werten anderer GET-Felder, gesteuert werden kann. Im folgenden bedeutet die Eingabe von "J", da. der Kunde eine Versicherung besitzt und somit Informationen zur Versicherung eingegeben werden k.nnen. Anderenfalls werden das zweite und dritte Feld .bersprungen und das Feld Unfall aufgerufen.

```
@ 10, 10 GET Versichert PICTURE "Y"   && J -> .T.
@ 11, 10 GET VNummer    WHEN Versichert
@ 12, 10 GET Agentur    WHEN Versichert
@ 13, 10 GET Unfall     PICTURE "Y"
READ
```

. Das folgende GET greift auf ein Datenbankfeld in einem anderen SELECT-Bereich zu:

```
USE Rechnung NEW
APPEND BLANK
USE Lager NEW
@ 1, 1 GET Lager->KundNr
READ
```

. Dieses Beispiel demonstriert die @K-Funktion. So kann ein Wert vorgegeben werden, der jedoch beim ersten Tastendruck (ausgenommen Cursor-Tasten oder [Return]) gel.scht wird:

```
LOCAL cDatei := "Rechnung"
@ 1, 1 SAY "Bitte Dateinamen eingeben:" GET cDatei PICTURE "@K"
READ
```

Dateien

Bibliothek ist CLIPPER.LIB

@...PROMPT Gibt einen Men.punkt aus und definiert eine dazugeh.rige Meldung

---

Syntax

@ <nZeile>, <nSpalte> PROMPT <cMen.Punkt> [MESSAGE <cMeldung>]

Argumente

<nZeile> und <nSpalte> geben die Zeilen- und Spaltenkoordinaten f.r den auszugebenden Men.punkt an. Dabei kann die Zeilenangabe Werte von Null bis MAXROW() und die Spaltenangabe Werte von Null bis MAXCOL() annehmen.

<cMen.Punkt> ist die als Men.punkt auszugebende Zeichenkette.

<cMeldung> gibt eine Meldung an, die jedesmal angezeigt wird, wenn der zugeh.rige Men.punkt markiert ist.

Anwendung

@...PROMPT geh.rt zum Ausgabeteil des Light-bar-Men.systems von Clipper. Jeder @...PROMPT-Befehl gibt einen Men.punkt in der aktuellen Standard-Farbeinstellung aus und definiert eine dazugeh.rende Meldung, die in der durch SET MESSAGE angegebenen Zeile angezeigt wird. Die eigentliche Men.auswahl wird durch den Befehl MENU TO aktiviert. Die Angabe der Men.punkte kann in beliebiger Reihenfolge und Zusammensetzung von Zeilen- und Spaltenpositionen erfolgen. Diese Reihenfolge wird allerdings der Cursor-Steuerung bei der Auswahl mit MENU TO zugrundegelegt. Pro Men. k.nnen bis zu 32 Men.eintr.ge definiert werden.

Nach jedem @...PROMPT-Befehl steht der Cursor in der Spalte hinter dem letzten Zeichen des Men.punktes. ROW() und COL() werden entsprechend dieser Position aktualisiert. Dadurch k.nnen Sie .ber ROW() und COL() aneinander anschlie.ende Ausgabepositionen f.r die anderen Men.punkte angeben, die alle relativ vom ersten Men.punkt abh.ngen (s. Beispiel).

Beispiele

Dieses Beispiel gibt ein Light-bar-Men. in der ersten Bildschirmzeile aus. Die jeweils zugeh.rigen Meldungen werden in der n.chsten Zeile angezeigt. Sobald der Anwender [Return] dr.ckt, wird die Position des gew.hlten Punktes in der Liste aller Men.punkte an die Variable nWahl .bergeben.

```
LOCAL nWahl := 1
SET WRAP ON
SET MESSAGE TO 2
@ 1, 3 PROMPT "Datei" MESSAGE ".ffnung einer Datendatei"
@ ROW(), COL() + 2 PROMPT "Bearbeitung" MESSAGE;
   "Bearbeitung des aktuellen Datensatzes"
MENU TO nWahl
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

ACLONE() Dupliziert ein verschachteltes oder mehrdimensionales Array

---

Syntax

ACLONE(<aQuelle>) ->aDuplikat

Parameter

<aQuelle> ist das Array, welches dupliziert werden soll.

R.ckgabe

ACLONE() liefert ein Duplikat von <aQuelle>.

Anwendung

ACLONE() ist eine Arrayfunktion, welche ein komplettes Duplikat von <aQuelle> erzeugt. Wenn <aQuelle> Subarrays enth.lt, erzeugt ACLONE() entsprechende neue Subarrays und f.llt sie mit Kopien der Werte, die auch in <aQuelle> enthalten sind. ACLONE() ist vergleichbar mit ACOPY(), jedoch werden bei ACOPY() keine verschachtelten Arrays dupliziert.

Beispiele

Dieses Beispiel erzeugt ein Array und dupliziert es mit ACLONE(). Das erste Array wird ver.ndert und das zweite bleibt unbeeinflu.t:

```
LOCAL aEins, aZwei
aEins := { 1, 2, 3 }          // aEins: {  1, 2, 3 }
aZwei := ACLONE(aEins)        // aZwei: {  1, 2, 3 }
aEins[1] := 99                // aEins: { 99, 2, 3 }
                              // aZwei bleibt { 1, 2, 3 }
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

@...SAY Gibt Daten auf dem Bildschirm oder dem Drucker an der angegebenen Zeilen- und Spaltenposition aus

---

Syntax

@ <nZeile>, <nSpalte> SAY <Ausdruck> [PICTURE <cSayPicture>]

Argumente

<nZeile> und <nSpalte> geben die Zeilen- und Spaltenkoordinaten f.r die Ausgabe an. Dabei kann die Zeilenangabe Werte von Null bis einschlie.lich MAXROW() annehmen, wenn SET DEVICE TO SCREEN eingestellt worden ist. Falls SET DEVICE TO PRINTER eingestellt ist, liegt die Obergrenze bei 32.766. Entsprechend kann die Spaltenangabe von Null bis MAXCOL(), also 32.766 reichen.

SAY <Ausdruck> gibt das Ergebnis eines Zeichen-, Datums-, logischen oder numerischen Ausdrucks auf der aktuellen DEVICE-Ausgabeeinheit aus.

PICTURE <cSayPicture> kontrolliert die Formatierung der Ausgabe von <Ausdruck>. Clipper stellt zwei M.glichkeiten zur Formatierung zur Verf.gung: Funktionen und Schablonen. Funktionen betreffen die Formatierung des kompletten SAY <Ausdrucks>, w.hrend Schablonen nur jeweils Zeichen f.r Zeichen von <Ausdruck> formatieren.

Anwendung

@...SAY ist ein Full-screen-Befehl, der die Resultate von <Ausdruck> entweder auf dem Bildschirm oder dem Drucker an den angegebenen Zeilen- und Spaltenpositionen ausgibt. Zus.tzlich kann die Ausgabe mit Hilfe der PICTURE-Klausel formatiert werden. @...SAY dient der Erstellung von Dateneingabemasken bzw. Berichten zur Ausgabe auf dem Drucker oder Bildschirm.

Bei Ausf.hrung von @...SAY wird das Ergebnis von <Ausdruck> an die augenblicklich definierte DEVICE-Ausgabeeinheit gesendet. Letztere kann entweder der Bildschirm oder der Drucker sein. Anders als bei Konsolen-Befehlen wird die Ausgabe auf dem Drucker nicht zus.tzlich auf dem Bildschirm ausgegeben. Der Befehl SET CONSOLE beeinflu.t die Bildschirmausgabe durch @...SAY nicht.

Bei Ausgabe auf der Default-Ausgabeeinheit Bildschirm (SCREEN) positioniert @...SAY den Cursor auf diejenige Spalte, die nach dem letzten dargestellten Zeichen folgt. ROW() und COL() werden entsprechend dieser Position aktualisiert. Ausgaben, die .ber die durch MAXROW() und MAXCOL() bestimmten Grenzen hinausgehen, werden abgeschnitten. Der Cursor befindet sich dann au.erhalb des sichtbaren Bildschirms. Die Darstellung durch @...SAY erfolgt immer in der Farbe standard. Weitere Informationen zur Einstellung der Bildschirmfarben finden Sie in diesem Handbuch bei SETCOLOR() im Kapitel Funktionen.

Bei Ausgabe auf dem Drucker beginnt die Darstellung ab der angef.hrten Position
<nZeile>, <nSpalte>. Ist der Druckrand .ber SET MARGIN TO auf einen Wert gr..er Null eingestellt, so wird dieser Wert zuerst zu <nSpalte> hinzuaddiert. @...SAY positioniert den Druckkopf auf die dem letzten ausgedruckten Zeichen folgende Spalte. PROW() und PCOL() werden entsprechend dieser Position aktualisiert. Der @...SAY-Befehl unterscheidet sich bei der Ausgabe auf dem Drucker in einem Punkt von der Bildschirmausgabe. Wenn die Ausgabe auf einer Zeilen- oder Spaltenposition vorgenommen werden soll, die vor der aktuellen Position von PROW() bzw. PCOL() liegt, so kommt es zu folgendem Verhalten:

. Ist die aktuelle Zeile <nZeile> kleiner als PROW(), so wird automatisch ein EJECT (CHR(12)) gefolgt von <nZeile> Zeilenvorsch.ben (CHR(10)) gesendet. Dadurch wird der Druckkopf auf der n.chsten Seite in die gleiche Zeile <nZeile> plaziert.

. Ist die aktuelle Spalte <nSpalte> inklusive des Wertes von SET MARGIN

TO kleiner als PCOL(), so wird automatisch ein Carriage Return (CHR(13)) gefolgt von <nSpalte> Leerzeichen (CHR(32)) gesendet. Dadurch wird der Druckkopf auf die entsprechende Spalte <nSpalte> plaziert.

Um dieses Verhalten zu umgehen, k.nnen Sie die Werte f.r PROW() und PCOL() durch SETPRC() ver.ndern oder zur.cksetzen. Auf diese Weise ist es z.B. m.glich, Steuerzeichen an den Drucker zu senden, ohne da. diese die Werte von PROW() und PCOL() beeinflussen werden. Weitere Informationen dazu finden Sie in diesem Handbuch im Kapitel Funktionen unter SETPRC().

Bei Ausgabe auf dem Drucker (SET DEVICE TO PRINTER) k.nnen @...SAY-Befehle in eine Datei umgelenkt werden. Dies erfolgt mit dem Befehl SET PRINTER TO <xcDatei>.

Eine Formatierung der Ausgabe eines @...SAY-Befehles kann, wie bereits erw.hnt, mit Hilfe der PICTURE-Klausel <cSayPicture> erfolgen. Das Ergebnis ist dasselbe wie bei Verwendung der Funktion TRANSFORM(). Der String <cSayPicture> kann aus einer Funktion und/oder Schablone bestehen. Eine PICTURE-Funktion beeinflu.t die @...SAY-Darstellung als Ganzes. Eine PICTURE-Schablone bestimmt zum einen die L.nge der @...SAY-Ausgabe und zum anderen die Formatierungsregel f.r jede einzelne Position der Ausgabe.

. Formatierungsfunktion: Eine PICTURE-Formatierungsfunktion spezifiziert eine bestimmte Formatierung oder G.ltigkeitspr.fung f.r das ganze Feld, anstatt f.r ein einzelnes Zeichen innerhalb des Feldes. Eine solche Formatierungsfunktion beginnt immer mit dem Zeichen @, gefolgt von einem oder mehreren Buchstaben, von denen wiederum jeder eine besondere Bedeutung hat (s. folgende Tabelle). Wird eine Formatierungsfunktion angegeben, so mu. das @-Zeichen immer das erste innerhalb des PICTURE-Strings sein und innerhalb der Formatierungsfunktion d.rfen keine Leerzeichen enthalten sein. Eine Formatierungsfunktion kann allein, aber auch gemeinsam mit einer PICTURE-Schablone angegeben werden. Werden beide Arten verwendet, so mu. die Formatierungsfunktion zuerst angegeben sein und beide m.ssen durch ein Leerzeichen voneinander getrennt werden.

Tabelle 4-3: SAY PICTURE Formatierungs-Funktionen

Funkt.	Typ	Auswirkung
B	N	Zahlen werden nach links ausgerichtet
C	N	Nach positiven Zahlen CR (Credit) anzeigen
D	D,N	Datum im SET DATE-Format anzeigen
E	D,N	Datum und Zahlen in (E)urop.ischer Schreibweise anzeigen
R	C	Zeichen, die keine Schablonen-Zeichen sind, werden in die Anzeige eingef.gt
X	N	Nach negativen Zahlen DB (Debit) anzeigen
Z	N	Anstelle f.hrender Nullen Leerzeichen anzeigen
(	N	Negative Zahlen werden einschlie.lich f.hrender Leerstellen in runden Klammern angezeigt
)	N	Negative Zahlen werden ohne f.hrende Leerstellen in runden Klammern angezeigt
!	C	Alle Kleinbuchstaben werden in Gro.buchstaben umgewandelt (UPPER CASE)

. Formatierungsschablone: Eine PICTURE-Formatierungsschablone gibt die Formatierungen und Regeln zur G.ltigkeitspr.fung f.r jedes einzelne Zeichen des Eingabefeldes an. Dabei sind eine Reihe von Schablonenzeichen mit besonderer Bedeutung definiert (s. folgende Tabelle). Jede Position in der Formatierungsschablone korrespondiert genau mit einer Stelle im aktiven Eingabefeld. Zeichen, die in der Formatierungsschablone keine besondere Bedeutung haben, werden ohne Ver.nderung oder Interpretation angezeigt. Wenn gleichzeitig die PICTURE-Funktion @R verwendet wird, werden diese Zeichen zwischen denen der Eingabevariablen eingef.gt, angezeigt und automatisch wieder gel.scht, wenn die ver.nderte Eingabe wieder <idVar> zugewiesen wird. Anderenfalls .berschreiben solche Zeichen die korrespondierenden Daten und beeinflussen so letztendlich auch den Inhalt von <idVar>. Eine

Formatierungsschablone kann allein, aber auch gemeinsam mit einer PICTURE-Funktion angegeben werden. Werden beide Arten verwendet, mu. die Formatierungsfunktion zuerst angegeben sein, und beide m.ssen durch ein Leerzeichen voneinander getrennt werden.

Tabelle 4-4: SAY PICTURE Schablonen-Symbole

Schablonen	Auswirkung
A,N,X,9,#	Zeigt Ziffern f.r jeden Datentyp
L	Zeigt logische Werte als "T" oder "F"
Y	Zeigt logische Werte als "J" oder "N"
!	Wandelt Klein- in Gro.buchstaben um
$	Zeigt bei numerischen Datentypen anstelle eines f.hrenden Leerzeichens ein $-Zeichen an
*	Zeigt bei numerischen Datentypen anstelle eines f.hrenden Leerzeichens das Zeichen * an
.	Zeigt einen Dezimalpunkt an
,	Zeigt ein Komma an

Beispiele

. Das folgende Beispiel demonstriert den Einsatz von @...SAY mit einer PICTURE-Klausel zur Formatierung der Ausgabe:

```
nNettoUmsatz := 7125.50
nNettoVerlust := -125.50
cTelefonNr := "021451030"
cName := "Willi Winzig"
//
@ 1, 1 SAY nNettoUmsatz PICTURE "@E 9,999.99" // Ergebnis: 7.125,50
@ 2, 1 SAY nNettoVerlust PICTURE "@E)"        // Ergebnis: (125,50)
@ 3, 1 SAY cTelefonNr PICTURE "@R (9999) 99999" // Ergebnis: (0214) 51030
@ 4, 1 SAY cName PICTURE "@!"                 // Ergebnis: WILLI WINZIG
```

. Dieses Beispiel ist ein einfaches Label-Druckprogramm. Dabei wird die Ausgabe .ber SET DEVICE auf den Drucker umgelenkt und automatische EJECTs werden durch SETPRC() unterdr.ckt:

```
USE Verkauf INDEX Verkauf NEW
SET DEVICE TO PRINTER
DO WHILE .NOT. EOF()                    // Drucke alle Datens.tze
  @ 2, 5 SAY RTRIM(Vorname) + " " + Nachname
  @ 3, 5 SAY Strasse
  @ 4, 5 SAY PLZ + " " + Ort
  @ 6, 1 SAY SPACE(1)                   // Springe zum Labelende
  SETPRC(0,0)                           // Unterdr.cke Seitenvorschub
  SKIP                                  // N.chster Datensatz
ENDDO
SET DEVICE TO SCREEN
CLOSE Verkauf
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

@...TO Zeichnet eine Box mit einfachen oder doppelten Linien

Syntax

@ <nOben>, <nLinks> TO <nUnten>, <nRechts> [DOUBLE]

Argumente

<nOben>, <nLinks>, <nUnten> und <nRechts> geben die Koordinaten der Box an. @...TO kann mit Zeilenwerten von 0 bis MAXROW() und Spaltenwerten von 0 bis MAXCOL() aufgerufen werden. Liegt die rechte untere Ecke au.erhalb des Bildschirmbereiches, so endet das Zeichnen der Box bei MAXROW(), MAXCOL().

Bei Angabe von DOUBLE wird die Box mit Doppellinien, ansonsten mit einfachen Linien gezeichnet.

Anwendung

@...TO zeichnet eine Box mit einer einfachen oder doppelten Linie auf den Bildschirm. Wenn <nOben> und <nUnten> gleich sind, wird eine horizontale Linie gezeichnet. Sollten <nLinks> und <nRechts> gleich sein, wird eine vertikale Linie gezeichnet.
Nach dem Aufbau der Box steht der Cursor in der linken oberen Ecke der Box auf den Koordinaten <nOben>+1 und <nLinks>+1. ROW() und COL() werden entsprechend aktualisiert.

@...TO ist dem @...BOX-Befehl .hnlich, mit der Ausnahme, da. Sie bei @...BOX die einzelnen Rahmenzeichen und das F.llzeichen angeben k.nnen. Wir empfehlen die Verwendung von @...TO aus Gr.nden der Portabilit.t, weil dieser Befehl nicht von der Verwendung bestimmter Semigrafikzeichen abh.ngig ist.

Beispiele

Im folgenden Beispiel wird ein Bereich des Bildschirms gel.scht und dann eine Doppellinien-Box gleicher Gr..e gezeichnet:

```
@ 10, 10 CLEAR TO 20, 40
@ 10, 10 TO 20, 40 DOUBLE
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

? | ?? Gibt einen oder mehrere Werte auf der gew.hlten Ausgabeeinheit aus

---

Syntax

? | ?? [<AusdrucksListe>]

Argumente

<AusdrucksListe> ist eine Liste der auszugebenden Werte und kann eine Kombination beliebiger Datentypen einschlie.lich Memo-Feldern sein.

Ist kein Argument angegeben, dann wird auf der Ausgabeeinheit bei Verwendung des ?-Befehles ein Carriage Return/Line Feed ausgegeben. Ein ??-Befehl ohne Argument wird ignoriert.

Beschreibung

? und ?? stellen Konsolen-Befehle dar, welche die Ergebnisse von einem oder mehreren Ausdr.cken auf der eingestellten Ausgabeeinheit (Konsole) ausgeben. Dabei trennt ein Leerzeichen die Ergebnisse voneinander. Diese beiden Befehle sind Synonyme f.r die Funktionen QOUT() und QQOUT(). Obwohl sie funktionell sehr .hnlich sind, unterscheiden sich ? und ?? geringf.gig voneinander. ? sendet vor Ausgabe der Ergebnisse der Ausdrucksliste ein Carriage Return/Line Feed an die Konsole. ?? gibt die Resultate an der aktuellen Cursor- oder Druckkopf-Position aus. Durch Verwendung von aufeinanderfolgenden ??-Befehlen k.nnen so mehrere verschiedene Ausgaben in der gleichen Zeile erfolgen.

Nach Anwendung eines ?- oder ??-Befehles befindet sich der Cursor bzw. die Druckkopf-Position immer hinter dem zuletzt ausgegebenen Zeichen. Wenn SET PRINTER OFF eingestellt ist, werden ROW() und COL() entsprechend der neuen Cursor-Position aktualisiert. Wenn SET PRINTER ON eingestellt ist, werden hingegen PROW() und PCOL() entsprechend der neuen Druckkopf-Position ver.ndert.

Wird durch einen ?- oder ??-Befehl das Ende einer Zeile erreicht, so erfolgt ein Umbruch in die n.chste Zeile. Bei Erreichen des unteren Bildschirmendes wird der gesamte Bildschirm um eine Zeile heraufgeschoben (Scrolling).

Die Ausgaben .ber ? und ?? k.nnen gleichzeitig auch auf einem Drucker erfolgen, wenn zuvor der Befehl SET PRINTER ON verwendet wird. Die gleichzeitige Ausgabe in eine Textdatei kann erfolgen, indem SET ALTERNATE TO <xcDatei> zur Erzeugung der Textdatei und danach SET ALTERNATE ON f.r den Beginn der Ausgabe in diese Datei angegeben wird. Wie bei den anderen Konsolen-Befehlen kann auch hier die Ausgabe auf den Bildschirm .ber SET CONSOLE OFF ausgeschaltet werden, ohne da. die zuvor beschriebenen Umlenkungen dadurch betroffen werden.

Um die als Argumente angegebenen Ausdr.cke zu formatieren, kann sehr gut die Funktion TRANSFORM() oder aber auch eine UDF verwendet werden. Wenn Sie einen der Werte auf eine bestimmte Spaltenposition ausrichten wollen, k.nnen Sie dies .ber eine der PAD()-Funktionen erreichen. Damit kann eine Zeichenkette durch Auff.llen mit Leerzeichen linksgerichtet, rechtsgerichtet oder zentriert ausgegeben werden, wie Sie im folgenden Beispiel sehen k.nnen.

Beispiele

Das folgende Beispiel gibt unter Benutzung des ?- und ??-Befehles alle Datens.tze einer Datenbank-Datei auf einen Drucker aus. Dabei wird die Funktion PADR() eingesetzt, um die Daten rechts mit Leerzeichen aufzuf.llen und so die Spalten entsprechend auszurichten:

```
LOCAL nSeite := 0, nZeile := 99
USE Verkauf INDEX Verkauf NEW
SET PRINTER ON
SET CONSOLE OFF
DO WHILE .NOT. EOF()
  IF nZeile > 55
```

```
    IF nSeite <> 0
      EJECT
    ENDIF
    ? PADR("Seite", LTRIM(STR(nSeite++)), 72)
    ?? DTOC(DATE())
    ?
    ?
    ? PADC("Verkaufs.bersicht", 79)
    ?
    nZeile := 5
  ENDIF
  ? Name, Adresse, PLZ, PADR(RTRIM(Stadt), 20)
  nZeile++
  SKIP
ENDDO
SET CONSOLE ON
SET PRINTER OFF
CLOSE Verkauf
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

ACCEPT Tastatureingabe in eine String-Variable

Syntax

ACCEPT [<cPromptAusdruck>] TO <idVar>

Argumente

<cPromptAusdruck> gibt optional einen Prompt-Text an, der vor dem Eingabebereich angezeigt wird. Dieser Prompt-Text kann ein Ausdruck beliebigen Datentyps sein.

<idVar> gibt die Variable an, der die Tastatureingaben zugewiesen werden sollen. Wenn diese Variable nicht existiert oder nicht sichtbar ist, wird eine PRIVATE-Variable angelegt.

Anwendung

ACCEPT ist ein Konsolen-Befehl, der Eingaben von der Tastatur entgegennimmt und diese einer Variablen zuweist. Bei der Ausf.hrung von ACCEPT wird zuerst ein Carriage Return/Line Feed und dann der Prompt ausgegeben. Danach werden die Tastatureingaben ab der ersten Position hinter dem Prompt entgegengenommen. Bis zu 255 Zeichen k.nnen dabei durch den Anwender eingegeben werden. Wird bei einer Eingabe der rechte Rand entsprechend MAXCOL() erreicht, so wandert der Cursor in die n.chste Zeile.

ACCEPT unterst.tzt nur zwei spezielle Tasten, Backspace und [Return], nicht jedoch [Esc]. ### l.scht die jeweils letzte Eingabe, w.hrend [Return] den Eingabemodus beendet. Wurden keine Eingaben vorgenommen, so weist ACCEPT der Variablen <idVar> einen Nullstring zu.

Beispiele

Dieses Beispiel demonstriert, wie .ber ACCEPT eine Benutzer-Eingabe realisiert werden kann:

```
LOCAL cVar
ACCEPT "Bitte Wert eingeben: " TO cVar
IF cVar == ""
  ? "Keine Benutzer-Eingabe!"
ELSE
  ? "Benutzer-Eingabe:", cVar
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

APPEND BLANK F.gt der selektierten Datenbank einen neuen Satz hinzu

---

Syntax

APPEND BLANK

Anwendung

APPEND BLANK ist ein Datenbankbefehl, der einen neuen Datensatz an die aktuelle Datenbank anh.ngt und den Datensatzzeiger auf diesen neuen Satz positioniert. Die Inhalte der einzelnen Datenbankfelder werden jeweils mit dem entsprechenden leeren Datentyp initialisiert: Zeichenfelder enthalten Leerstellen, numerische Felder beinhalten Null, logische Felder enthalten false (.F.), Datumsfelder enthalten ein CTOD(""), und Memo-Felder sind ganz einfach leer.

Wenn bei der Arbeit im Netzwerk die aktuelle Datenbank im shared-Modus benutzt wird, versucht APPEND BLANK, den neuen Datensatz anzuh.ngen und direkt zu sperren. Hat aber bereits ein anderer Anwender die Datenbank komplett mit FLOCK() oder LASTREC()+1 mit RLOCK() gesperrt, so gibt NETERR() true (.T.) zur.ck. Ein neu angehangener Datensatz bleibt so lange gesperrt, bis Sie ihn mit UNLOCK freigeben oder einen anderen Datensatz sperren. APPEND BLANK hebt ein FLOCK() des aktuellen Benutzers nicht auf.

Beispiele

In diesem Beispiel wird versucht, einen Datensatz an eine shared-Datenbank anzuh.ngen, wobei .ber NETERR() der Erfolg dieser Aktion abgefragt wird:

```
USE Verkauf SHARED NEW
.
.<Programmcode>...
.
APPEND BLANK
IF .NOT. NETERR()
.
.<Aktualisierung des leeren Datensatzes>...
.
ELSE
  ? "Datensatz konnte nicht angeh.ngt werden!"
  BREAK
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

ACOPY() Kopiert Elemente von einem Array in ein anderes

Syntax

ACOPY(<aQuelle>, <aZiel>,
[<nStart>], [<nAnzahl>], [<nZielPosition>]) -> aZiel

Parameter

<aQuelle> ist das Array, aus dem kopiert werden soll.

<aZiel> ist das Array, in das kopiert werden soll.

<nStart> gibt die Position des Elements von <aQuelle> an, bei dem mit dem Kopiervorgang begonnen wird. Ohne Parameter wird beim ersten Element begonnen.

<nAnzahl> ist die Anzahl der zu kopierenden Elemente ab der Position <nStart>. Wird der Parameter nicht angegeben, so wird bis zum letzten Element kopiert.

<nZielPosition> ist die Position des Elements von <aZiel>, welche das erste zu kopierende Element empfangen soll. Der Vorgabewert ist das erste Element.

R.ckgabe

ACOPY() liefert eine Referenz auf <aZiel>.

Anwendung

ACOPY() ist eine Arrayfunktion, welche Elemente von einem Array in ein anderes kopiert. Das Zielarray <aZiel> mu. bereits existieren und in den Dimensionen so bemessen sein, da. es die zu kopierenden Elemente aufnehmen kann. Werden mehr Elemente kopiert, als das Zielarray aufnehmen kann, gehen die .berz.hligen Elemente verloren.

ACOPY() kopiert Elemente aller Datentypen inclusive der Datentypen NIL, Codeblock und Subarrays. Ist ein Element von <aQuelle> ein Subarray, so wird das korrespondierende Element von <aZiel> die Referenz auf das gleiche Subarray enthalten. ACOPY() erstellt keine komplette Kopie des Subarrays. Dies kann .ber die Funktion ACLONE() erreicht werden.

Beispiele

Dieses Beispiel erzeugt zwei Arrays, jedes mit Werten gef.llt. Die ersten zwei Elemente des Quellarrays werden anschlie.end in ein Zielarray kopiert:

```
LOCAL nAnzahl := 2, nStart := 1
LOCAL aEins := { 1, 1, 1 }, aZwei := { 2, 2, 2 }
ACOPY(aEins, aZwei, nStart, nAnzahl)        //Resultat aZwei: { 1, 1, 2 }
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

APPEND FROM Importiert Datens.tze aus .DBF- oder ASCII-Dateien

---

Syntax

APPEND FROM <xcDatei>
[FIELDS <idFelderListe> [<Wirkungsbereich>]
[WHILE <lBedingung>] [FOR <lBedingung>]
[SDF | DELIMITED [WITH BLANK | <xcBegrenzer>]]

Argumente

FROM <xcDatei> gibt den Namen der Quelldatei an. Der Name kann entweder als Literal oder als eingeklammerter Zeichenausdruck (erweiterter Ausdruck) angegeben werden. Ist kein Dateityp spezifiziert, wird .DBF als Vorgabe verwendet. Bei Verwendung von SDF oder DELIMITED wird .TXT als Vorgabetyp f.r die Quelldatei angenommen, au.er Sie geben eine andere Dateierweiterung an.

FIELDS <idFelderListe> enth.lt eine Liste der Felder, die von <xcDatei> .bernommen werden sollen. Die Vorgabe umfa.t alle Felder.

<Wirkungsbereich> spezifiziert den Teil, der von der Quell-Datenbank .bernommen und angeh.ngt werden soll. Mit NEXT <n> werden die ersten <n> Datens.tze angeh.ngt. Durch RECORD <n> wird nur der einzelne Datensatz Nummer <n> von <xcDatei> .bernommen. Die Vorgabe umfa.t alle Datens.tze aus <xcDatei> (ALL).

WHILE <lBedingung> spezifiziert die Gruppe der Datens.tze in <xcDatei>, die sich von dem an der aktuellen Position vorhandenen bis zum ersten, die Bedingung nicht erf.llenden Datensatz erstreckt.

FOR <lBedingung> selektiert alle Datens.tze innerhalb des <Wirkungsbereiches> in <xcDatei>, welche die angef.hrte Bedingung erf.llen.

SDF kennzeichnet ein System Data Format ASCII File (also eine ASCII-Datei im Systemdatenformat). Jeder Datensatz besitzt eine feste, durch die Struktur der selektierten .DBF-Datei vorgegebene L.nge und ist durch ein Carriage Return/Line Feed begrenzt. Als Dateiende-Marke wird [Strg]+[B] (1A hex) verwendet.

DELIMITED kennzeichnet eine ASCII-Textdatei, in der die Felder durch Kommata voneinander getrennt und Zeichenfelder in Anf.hrungszeichen (das sind die Vorgabe-Begrenzungszeichen) eingeschlossen sind. Die Begrenzungszeichen sind nicht zwingend notwendig. Clipper h.ngt auch Zeichenfelder korrekt an, die nicht in Begrenzern eingeschlossen sind. Felder und Datens.tze d.rfen in der L.nge variieren. Datens.tze werden durch Carriage Return/Line Feed voneinander getrennt. Als Dateiende-Marke wird [Strg]+[A] (1A hex) verwendet.

DELIMITED WITH BLANK kennzeichnet eine ASCII-Textdatei, in der die Felder durch ein Leerzeichen voneinander getrennt und Zeichenfelder nicht in Begrenzern eingeschlossen sind.

DELIMITED WITH <xcBegrenzer> kennzeichnet eine ASCII-Textdatei, in der die Zeichenfelder durch das angegebene Begrenzungszeichen voneinander getrennt sind. <xcBegrenzer> kann entweder als Literal oder als eingeklammerter Zeichenausdruck angegeben werden.

Achtung

Wird die Klausel DELIMITED WITH in einer Befehlszeile mit COPY TO oder APPEND FROM angegeben, so mu. diese Klausel als letzte Klausel aufgef.hrt werden.

Anwendung

APPEND FROM h.ngt an die selektierte Datenbank Datens.tze aus einer ASCII-Textdatei oder einer anderen Datenbank an. Es werden dabei nur Felder gleichen Namens und Typs angeh.ngt. Felder mit gleichem Namen m.ssen in der aktuellen Zieldatei und der Quelldatei <xcDatei> vom

gleichen Datentyp sein. Anderenfalls tritt ein Laufzeitfehler bei Aufruf des APPEND FROM-Befehles auf.

Wenn Sie mit APPEND FROM in einer Netzwerkumgebung arbeiten m.chten, mu. die aktuelle Datenbank nicht mit USE EXCLUSIVE ge.ffnet oder mit FLOCK() gesperrt worden sein. Falls beim Anh.ngen von einem Datensatz andere Benutzer im Netz auch auf die Datenbank zugreifen, regelt Clipper dies automatisch.

Nach Aufruf des APPEND FROM-Befehles versucht Clipper, <xcDatei> mit den Attributen shared und read-only zu .ffnen. Wurde der Zugriff verweigert, so bricht APPEND FROM mit einem Laufzeitfehler ab. Weitere Informationen zur Programmierung von Netzwerkanwendungen finden Sie im Handbuch Programmierung und Hilfsprogramme.

Hinweise

. Gel.schte Datens.tze: Als gel.scht gekennzeichnete Datens.tze werden aus <xcDatei> .bernommen und angehangen, ohne da. die L.schmarke zur.ckgesetzt wird. Wurde allerdings SET DELETED auf ON eingestellt, so h.ngt APPEND FROM keinen der in <xcDatei> als gel.scht gekennzeichneten Datens.tze an.

. Unterschiedliche Feldl.ngen: Ist die L.nge eines Zeichenfeldes in der aktuellen Datenbank gr..er als die des korrespondierenden Datenbankfeldes in <xcDatei>, so werden die ankommenden Daten f.r dieses Feld mit Leerstellen aufgef.llt. Ist das Zeichenfeld kleiner als das Datenbankfeld in <xcDatei>, so werden die angehangenen Daten abgeschnitten, soweit sie l.nger als das Zeichenfeld der aktuellen Datenbank sind. Vorsicht ist bei numerischen Feldern geboten: Besitzt ein numerisches Feld der Quelldatei <xcDatei> mehr Ziffern als das korrespondierende aktuelle Feld der Zieldatei, so tritt ein Laufzeitfehler auf.

Beispiele

. Das folgende Beispiel demonstriert den Einsatz von APPEND FROM mit einer Felderliste und Bedingung:

```
USE Verkauf NEW
APPEND FROM Filiale FIELDS Filiale, Verkaeufer, Umsatz;
  FOR Filiale = 100
```

. Dieses Beispiel zeigt, wie unter Verwendung eines <Wirkungsbereiches> ein bestimmter Datensatz aus einer anderen Datenbank importiert werden kann:

```
APPEND RECORD 5 FROM Temp
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

AVERAGE Berechnet den Mittelwert f.r numerische Ausdr.cke der selektierten Datenbank

---

Syntax

AVERAGE <nAusdrucksListe> TO <idVarListe>
[<Wirkungsbereich>] [WHILE <lBedingung>] [FOR <lBedingung>]

Argumente

<nAusdrucksListe> ist eine Liste von numerischen Ausdr.cken, deren Mittelwert aus allen verarbeiteten Datens.tze berechnet werden soll.

<idVarListe> gibt die Speichervariablen an, denen die Ergebnisse der Mittelwertberechnungen zugewiesen werden sollen. Nicht existierende oder nicht sichtbare Variablen werden als PRIVATE-Variable angelegt. <idVarListe> mu. genau die gleiche Anzahl an Elementen enthalten wie <nAusdrucksListe>.

<Wirkungsbereich> spezifiziert den Teil der Datens.tze in der aktuellen Datenbank, der von AVERAGE ber.cksichtigt werden soll. Die Vorgabe umfa.t alle Datens.tze (ALL).

WHILE <lBedingung> spezifiziert die Gruppe von Datens.tzen, die sich vom aktuellen bis zum ersten, die Bedingung nicht erf.llenden Datensatz erstreckt.

FOR <lBedingung> selektiert alle Datens.tze innerhalb des <Wirkungsbereiches>, welche die angef.hrte Bedingung erf.llen. Diese Datens.tze werden von AVERAGE ber.cksichtigt.

Anwendung

AVERAGE berechnet den Mittelwert von einem oder mehreren numerischen Ausdr.cken f.r einen bestimmten <Wirkungsbereich> innerhalb der aktuellen Datenbank und weist die Ergebnisse Speichervariablen zu. Nullwerte werden von AVERAGE immer ber.cksichtigt, au.er Sie schlie.en sie durch die FOR-Bedingung aus.

Beispiele

. In diesem Beispiel wird der Mittelwert von einem einzelnen Datenbankfeld f.r einen Teil der Datens.tze aus der aktuellen Datenbank berechnet:

```
USE Verkauf NEW
AVERAGE Umsatz TO nMitUmsatz FOR Filiale = "100"
```

. Dieses Beispiel zeigt, wie das Durchschnittsdatum f.r eine Anzahl von Daten ermittelt werden kann:

```
AVERAGE (VerkDatum - CTOD("00.00.00") ;
  TO nMitTage FOR .NOT. EMPTY(VerkDatum)
dMitDatum = CTOD("00.00.00") + nMitTage
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

CALL* Ruft eine C- oder Assembler-Prozedur auf

Syntax

CALL <idProzedur> [WITH <AusdrucksListe>]

Argumente

<idProzedur> ist der Name der externen Prozedur, die .ber CALL aufgerufen werden soll.

<AusdrucksListe> ist eine optional anzugebende Liste von bis zu sieben Ausdr.cken jeglichen Datentyps, die an die externe Prozedur .bergeben werden.

Anwendung

CALL f.hrt eine separat, d.h. nicht mit Clipper erstellte Prozedur aus. Diese Prozedur mu. als FAR deklariert und mit einer FAR-R.ckgabe beendet werden. Die Parameter werden auf dem Stack abgelegt, wobei die C-Konventionen f.r Parameter.bergaben beachtet werden m.ssen. Jeder Parameter besteht aus einem vier Byte gro.en FAR-Zeiger, der auf den eigentlichen Parameterwert zeigt. Abweichend davon kann mit der WORD()-Funktion ein zwei Byte gro.er Bin.rwert .bergeben werden. Die Register DX:BX und ES:BX enthalten ebenfalls die ersten vier Bytes der Parameter-Informationen.

Die aufgerufene Prozedur mu. die Register BP, SS, SI, DI, ES und DS sichern und das Direction-Flag zur.cksetzen.

CALL ist ein Kompatibilit.ts-Befehl und daher nicht f.r den Einsatz in neuen Applikationen empfohlen. Er wird durch das Extend System ersetzt, das Funktionen zur Daten.bergabe an und Daten.bernahme von Clipper zur Verf.gung stellt. Weitere Informationen zum Extend System finden Sie im entsprechenden Kapitel des Handbuches Programmierung und Hilfsprogramme.

Hinweise

. Zeichenketten: Ein Zeichenargument wird als FAR-Zeiger auf einen nullterminierten String .bergeben (der String endet mit dem Byte 00 hex).

. Numerische Werte: Jedes numerische Argument wird als FAR-Zeiger auf eine acht Byte lange Flie.kommazahl im IEEE-Format .bergeben. Integerwerte werden mit der WORD()-Funktion .bergeben, die den Wert in eine zwei Byte lange bin.re Integerzahl umwandelt. Ein Integerwert wird direkt und nicht per Zeiger .bergeben. Die Funktion WORD() kann allerdings keine Werte gr..er als 32.767 und kleiner als -32.768 verarbeiten, da solche Werte nicht mehr eindeutig als 2-Byte-Integerzahl darstellbar sind.

. Datumswerte: Ein Datumsargument wird als FAR-Zeiger auf eine 4-Byte-Integerzahl (also vom Typ long) .bergeben, die eine Tageszahl entsprechend dem Julianischen Kalender darstellt.

. Logische Werte: Ein logisches Argument wird als FAR-Zeiger auf eine bin.re 2-Byte-Integerzahl .bergeben, die entweder 1 f.r true (.T.) oder 0 f.r false (.F.) enth.lt.

. Compilieren und linken: .ber CALL aufgerufene Programme m.ssen die folgenden Regeln beachten:

  Prozeduren m.ssen in einer .OBJ-Datei abgelegt sein, die das INTEL-8086-Relocatable-Object-File-Format einhalten.

  Prozeduren m.ssen die C-Konventionen f.r Aufrufe und Parameter.bergaben befolgen.

  Prozeduren m.ssen w.hrend des Linkprozesses zusammen mit der Library des Quell-Compilers f.r den Linker zug.nglich sein. Bei Prozeduren, die in einer anderen Sprache als Assembler geschrieben sind, werden in vielen F.llen zus.tzliche Runtime-Module ben.tigt. Die n.tigen Informationen finden Sie im jeweiligen Compiler-Handbuch.

. Microsoft C: Ab der Version 5.0 von Microsoft C beginnen Funktionsnamen beim Compilieren mit einem Unterstrich. Diese Funktionen m.ssen Sie demnach mit _<Funktionsname> aufrufen.

. dBASE III PLUS: Zur Umwandlung eines Lademoduls f.r dBASE III PLUS in ein Clipper-kompatibles Modul f.gen Sie folgende Anweisungen in Ihre .ASM-Datei ein:

```
PUBLIC <Prozedurname>
```

und

```
push ds
mov ds,dx
```

Achtung
Eine Ver.nderung von Parameterwerten kann falsche oder unerwartete Ergebnisse produzieren und sollte daher unter keinen Umst.nden vorgenommen werden.

Dateien
Bibliothek ist CLIPPER.LIB

See Also:

CANCEL* Beendet das laufende Programm

Syntax

CANCEL* | QUIT

Anwendung

QUIT und CANCEL beenden beide das laufende Programm, schlie.en alle offenen Dateien und .bergeben die Kontrolle zur.ck an das Betriebssystem. Beide Befehle k.nnen an jeder beliebigen Stelle eines Clipper-Programmes eingesetzt werden. Ein RETURN-Befehl, ausgef.hrt in der obersten Prozedurebene, hat die gleiche Wirkung.

Hinweise

. R.ckgabewert: Wird ein Clipper-Programm durch einen schwerwiegenden Fehler beendet, so wird der DOS-ERRORLEVEL auf 1 gesetzt. Wird das Programm hingegen normal beendet, so ist dieser R.ckgabewert 0 oder aber die letzte .ber ERRORLEVEL() vorgenommene Einstellung.

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

CLEAR ALL* Schlie.t alle Datenbanken und gibt alle PUBLIC- und PRIVATE-Variablen frei

---

Syntax

CLEAR ALL

Anwendung

CLEAR ALL l.scht alle PUBLIC- und PRIVATE-Variablen, schlie.t alle Datenbanken, damit zusammenh.ngende Index-, Alternate- und Memo-Dateien und selektiert den Arbeitsbereich 1. LOCAL- und STATIC-Variablen werden durch CLEAR ALL nicht beeinflu.t.

CLEAR ALL ist ein Kompatibilit.ts-Befehl und wird daher nicht f.r den Einsatz in neuen Applikationen empfohlen. Stattdessen sollten die f.r die jeweiligen Bereiche separat zur Verf.gung stehenden Befehle verwendet werden. Dateien in den verschiedenen Arbeitsbereichen k.nnen .ber den CLOSE-Befehl geschlossen werden. PUBLIC- und PRIVATE-Variablen k.nnen .ber den Befehl RELEASE gel.scht werden, obwohl das explizite L.schen von Variablen in der Regel nicht empfehlenswert ist. Weitere Informationen zur Sichtbarkeit und Existenz von Variablen finden Sie im Kapitel Grundkonzepte innerhalb dieses Handbuches.

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

CLEAR GETS Gibt alle GET-Objekte aus dem aktuellen GetList-Array frei

---

Syntax

CLEAR GETS

Anwendung

CLEAR GETS l.scht alle GET-Objekte im aktuellen und sichtbaren GetList-Array. Das aktuelle READ wird beendet, wenn CLEAR GETS aus einer SET KEY-Prozedur oder einer VALID-UDF heraus ausgef.hrt wird. Die L.schung der GET-Objekte erfolgt, indem der Variablen GetList ein leeres Array zugewiesen wird. Die Variable GetList enth.lt ein Array von GET-Objekten, die f.r nachfolgende READ-Befehle benutzt werden. GET-Objekte k.nnen au.erdem auf zwei andere Arten freigegeben werden: Einmal durch den Befehl CLEAR ohne den Zusatz SCREEN und zum anderen durch READ ohne die SAVE-Klausel.

CLEAR GETS wird normalerweise in zwei F.llen eingesetzt: Zum einen wird es benutzt, um ein READ aus einer SET KEY-Prozedur oder einer VALID-UDF heraus zu beenden. Der zweite Fall liegt vor, wenn Sie GET-Objekte aus dem GetList-Array l.schen m.chten, ohne vorher ein READ ausgef.hrt zu haben oder wenn Sie die GET-Objekte bereits mit READ SAVE gesichert haben.

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

CLEAR MEMORY Gibt alle PUBLIC- und PRIVATE-Variablen frei

Syntax

CLEAR MEMORY

Anwendung

CLEAR MEMORY l.scht sowohl alle PUBLIC- als auch PRIVATE-Speichervariablen aus der Speichervariablen-Tabelle. Dabei arbeitet CLEAR MEMORY anders als RELEASE ALL. RELEASE ALL l.scht die PUBLIC- und PRIVATE-Speichervariablen nicht wirklich aus dem Arbeitsspeicher, sondern weist allen Speicher-variablen, die auf der aktuellen Prozedurebene erzeugt worden sind, NIL zu. Der Aufruf von CLEAR MEMORY ist daher der einzige Weg, um alle aktuell vorhandenen PUBLIC- und PRIVATE-Speichervariablen aus dem Speicher zu entfernen. CLEAR MEMORY beeinflu.t jedoch keine LOCAL- und STATIC-Variablen.

Weitere Informationen .ber Variablen finden Sie in diesem Handbuch im Kapitel Grundkonzepte.

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

CLEAR SCREEN L.scht den Bildschirm und plaziert den Cursor auf die Position (0,0)

Syntax

CLEAR [SCREEN] | CLS

Argumente

SCREEN unterdr.ckt die automatische L.schung der GET-Objekte aus dem aktuellen und sichtbaren GetList-Array.

Anwendung

CLEAR ist ein Full-screen-Befehl, der den gesamten Bildschirm und alle ausstehenden GETs l.scht sowie den Cursor auf die Home-Position (0,0) setzt. Bei Angabe der SCREEN-Klausel werden GET-Objekte nicht gel.scht.

CLS ist ein Synonym f.r CLEAR SCREEN.

Hinweise

. SET KEY und VALID: Das aktuelle READ wird beendet, wenn Sie CLEAR bei der Eingabe in GET-Felder aus einer SET KEY-Prozedur oder einer VALID-UDF heraus ausf.hren. Sie k.nnen den Bildschirm ohne Eliminierung der GETs l.schen, indem Sie CLEAR SCREEN oder CLS verwenden.

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

CLEAR TYPEAHEAD L.scht den Tastatur-Eingabepuffer

Syntax

CLEAR TYPEAHEAD

Anwendung

CLEAR TYPEAHEAD ist ein Tastatur-Befehl, der alle noch im Clipper-Tastatur-Eingabepuffer befindlichen Tastencodes l.scht. Das ist z.B. dann wichtig, wenn der Anwender Eingaben vornehmen soll, die sich nur auf die aktuelle Prozedur oder UDF beziehen sollen und nicht von vorhergehenden Programmteilen noch .briggeblieben sind. Dies ist besonders bei den Anwenderfunktionen von ACHOICE() und DBEDIT() der Fall.

Der Tastatur-Eingabepuffer wird noch von zwei anderen Befehlen gel.scht, n.mlich von SET TYPEAHEAD und KEYBOARD.

Beispiele

In diesem Beispiel wird der Tastatur-Eingabepuffer vor dem Aufruf von DBEDIT() gel.scht. Dadurch wird sichergestellt, da. eventuell noch ausstehende Tasteneingaben nicht durch DBEDIT() verarbeitet werden.

CLEAR TYPEAHEAD
DBEDIT()

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

CLOSE Schlie.t eine bestimmte Gruppe von Datenbanken

Syntax

CLOSE [<idAlias> | ALL | ALTERNATE | DATABASES | FORMAT | INDEXES]

Argumente

<idAlias> gibt den Arbeitsbereich an, in dem alle Dateien geschlossen werden sollen.

ALL schlie.t alle Alternate-, Datenbank- und Indexdateien in allen Arbeitsbereichen. Au.erdem werden alle aktiven Filter, Verbindungen und Formatdefinitionen zur.ckgesetzt.

ALTERNATE schlie.t die aktuell ge.ffnete Alternate-Datei. Die gleiche Wirkung hat der Befehl SET ALTERNATE TO ohne Argument.

DATABASES schlie.t alle offenen Datenbank-, Memo- und zugeh.rige Indexdateien in allen Arbeitsbereichen. Au.erdem werden alle aktiven Filter und Verbindungen zur.ckgesetzt. Die aktiven Formatdefinitionen werden allerdings nicht beeinflu.t.

FORMAT l.scht das aktuelle Format. Die gleiche Wirkung hat der Befehl SET FORMAT TO ohne Argument.

INDEXES schlie.t alle offenen Indexdateien im aktuellen Arbeitsbereich.

Anwendung

CLOSE ist ein universeller Befehl, der, abh.ngig vom angegebenen Argument, verschiedene Clipper-Dateitypen schlie.t. Die Angabe von CLOSE ohne jeglichen Zusatz hat denselben Effekt wie USE ohne Argumente: Die aktuelle Datenbank und alle zugeh.rigen Dateien werden geschlossen.

In Clipper gibt es noch verschiedene andere Befehle, die Dateien schlie.en. Dazu geh.ren:

. QUIT
. CANCEL*
. RETURN von der obersten Prozedurebene aus
. CLEAR ALL*
. USE ohne Argument

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

COMMIT Schreibt die Clipper-Puffer aller Arbeitsbereiche auf Platte

---

Syntax

COMMIT

Anwendung

COMMIT ist ein Datenbankbefehl, der alle Clipper-Puffer physisch auf das aktuelle Datenlaufwerk schreibt. Dieser Schreibvorgang betrifft alle Arbeitsbereiche mit allen offenen Datenbank- und Indexdateien und ist ab DOS 3.3 verf.gbar. Bei allen DOS-Versionen kleiner als 3.3 gibt Clipper den Pufferinhalt nur an die DOS-Puffer weiter.

COMMIT f.hrt in einer Netzwerkumgebung alle Aktualisierungen der Datenbank- und ihrer zugeh.rigen Indexdateien durch, die f.r andere Netzwerkprozesse sichtbar und zug.nglich sind. Weitere Informationen zur Programmierung von Netzwerkanwendungen finden Sie im Handbuch Programmierung und Hilfsprogramme.

Beispiele

In diesem Beispiel wird COMMIT zur Ausl.sung eines Platten-Schreibvorganges benutzt, nachdem mehreren Datenbankfeldern die Werte von Speichervariablen zugewiesen worden sind:

```
USE Verkauf EXCLUSIVE NEW
MEMVAR->Name := Verkauf->Name
MEMVAR->Umsatz := Verkauf->Umsatz
//
@ 10, 10 GET MEMVAR->Name
@ 11, 10 GET MEMVAR->Umsatz
READ
//
IF UPDATED()
  APPEND BLANK
  REPLACE Verkauf->Name WITH MEMVAR->Name
  REPLACE Verkauf->Umsatz WITH MEMVAR->Umsatz
  COMMIT
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

CONTINUE Setzt ein vorheriges LOCATE fort

Syntax

CONTINUE

Anwendung

CONTINUE ist ein Datenbankbefehl, der im selektierten Arbeitsbereich vom aktuellen Datensatz an den n.chsten Datensatz sucht, der die Bedingung der zuletzt mit LOCATE ausgef.hrten Suche erf.llt. Die Suche wird beendet, wenn eine .bereinstimmung gefunden oder das Dateieende EOF() erreicht worden ist. Bei erfolgreicher Suche wird der gefundene Datensatz zum aktuellen Datensatz und FOUND() gibt true (.T.) zur.ck. Bei erfolgloser Suche gibt FOUND() false (.F.) zur.ck.

Jedem Arbeitsbereich kann eine aktive LOCATE-Bedingung zugeordnet sein, die in Clipper so lange g.ltig ist, bis eine neue LOCATE-Bedingung angegeben wird. Es existieren keine weiteren Befehle, die diese Bedingung aufheben.

Hinweise

. <Wirkungsbereich> und WHILE-Bedingung: Die urspr.nglich bei LOCATE angegebenen Argumente f.r den <Wirkungsbereich> und die WHILE-Bedingung werden von CONTINUE ignoriert. Nur eine urspr.nglich angegebene FOR-Bedingung wird ber.cksichtigt. Wenn Sie ein LOCATE unter Einbeziehung einer WHILE-Bedingung mit CONTINUE fortsetzen m.chten, m.ssen Sie zuerst SKIP ausf.hren und dann die originale LOCATE-Anweisung unter Angabe von REST f.r den <Wirkungsbereich> wiederholen.

Beispiele

. Das folgende Beispiel durchsucht Verkauf.DBF nach einem bestimmten Verk.ufer und zeigt die laufenden Ums.tze dieses Verk.ufers an:

```
LOCAL nLfdUmsatz := 0
USE Verkauf NEW
LOCATE FOR Verkauf->Verkaeufer = "1002"
DO WHILE FOUND()
  ? Verkauf->Verkaeufer, nLfdUmsatz += Verkauf->Umsatz
  CONTINUE
ENDDO
```

. Dieses Beispiel demonstriert den Einsatz von CONTINUE unter Ber.cksichtigung der urspr.nglichen WHILE-Bedingung von LOCATE:

```
LOCAL nLfdUmsatz := 0
USE Verkauf INDEX Verkauf NEW
SEEK "1002"
LOCATE REST WHILE Verkauf->Verkaeufer = "1002" FOR Verkauf->Umsatz >5000
DO WHILE FOUND()
  ? Verkauf->Verkaeufer, nLfdUmsatz += Verkauf->Umsatz
  SKIP
  LOCATE REST WHILE Verkauf->Verkaeufer = "1002" FOR Verkauf->Umsatz>5000
ENDDO
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

COPY FILE Kopiert eine Datei in eine neue Datei

Syntax

COPY FILE <xcQuellDatei> TO <xcZielDatei>

Argumente

<xcQuellDatei> gibt den Namen und die Erweiterung der zu kopierenden Datei an.

<xcZielDatei> gibt den Namen und die Erweiterung der neuen Zieldatei an.

Beide Argumente k.nnen als Literale oder eingeklammerte Zeichenausdr.cke angef.hrt werden. COPY FILE gibt keine Dateinamen-Erweiterung vor.

Anwendung

COPY FILE ist ein Datei-Befehl, der Dateien vom und in den Clipper-Vorgabepfad kopiert, au.er es werden andere Angaben f.r Laufwerk und Verzeichnis gemacht. Falls bereits eine Datei mit dem Namen der <xcZielDatei> existiert, wird diese Datei ohne Vorwarnung .berschrieben.

Beispiele

In diesem Beispiel wird eine Datei dupliziert und die korrekte Anlage der neuen Datei .berpr.ft:

```
COPY FILE Test.PRG TO Kopie.PRG
? FILE("Kopie.PRG")            // Ergebnis: .T.
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

COPY STRUCTURE Kopiert die Struktur der selektierten .DBF-Datei in eine neue .DBF-Datei

---

Syntax

COPY STRUCTURE [FIELDS <idFelderListe>] TO <xcDatenbank>

Argumente

FIELDS <idFelderListe> enth.lt eine Liste der Felder, die in die Struktur der neuen Datenbank kopiert werden sollen. Die Vorgabe umfa.t alle Felder.

TO <xcDatenbank> gibt den Namen der Zieldatei an. Der Name kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Die Vorgabe (default) f.r die Dateinamen-Erweiterung ist .DBF.

Anwendung

COPY STRUCTURE ist ein Datenbankbefehl, der eine leere Datenbank mit der Struktur der aktuellen Datenbank erzeugt. Falls bereits eine Datei mit dem Namen der <xcDatenbank> existiert, wird diese Datei ohne Vorwarnung .berschrieben.

COPY STRUCTURE wird haupts.chlich daf.r benutzt, die Datens.tze der aktuellen Datenbank in einer Datei mit gleicher Struktur zu archivieren. Eine Anwendung dieses Kommandos liegt in der Erzeugung einer tempor.ren Datei zur Dateneingabe.

Beispiele

In diesem Beispiel erzeugt COPY STRUCTURE eine tempor.re Datei. Nach der Dateneingabe durch den Benutzer wird die Hauptdatei mit den neuen Informationen aktualisiert.

```
LOCAL lWeiter := .F.
USE Verkauf NEW
COPY STRUCTURE TO Temp
USE Temp NEW
DO WHILE lWeiter
  APPEND BLANK
  @ 10, 10 GET Temp->Verkaeufer
  @ 11, 10 GET Temp->Umsatz
  READ
  IF UPDATED()
    SELECT Verkauf
    APPEND BLANK
    REPLACE Verkauf->Verkaeufer WITH Temp->Verkaeufer
    REPLACE Verkauf->Umsatz WITH Temp->Umsatz
    SELECT Temp
    ZAP
  ELSE
    lWeiter := .F.
  ENDIF
ENDDO
CLOSE DATABASES
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

COPY STRUCTURE EXTENDED Kopiert die Felddefinitionen der selektierten Datenbank in eine neue .DBF-Datei

---

Syntax

COPY STRUCTURE EXTENDED TO <xcExtendedDatenbank>

Argumente

<xcExtendedDatenbank> gibt den Namen der neuen Datenbank an, welche die Struktur der selektierten Datenbank als Datens.tze enthalten soll. Der Name kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden.

Anwendung

COPY STRUCTURE EXTENDED erzeugt eine Datenbank, deren Inhalt die Struktur der aktuellen Datenbank darstellt. Dabei wird f.r jedes Datenbankfeld ein Satz angelegt. Eine so erzeugte, bereits um die Struktur einer Datenbank erweiterte Struktur-Datei hat folgenden Aufbau:

Tabelle 4-5: Struktur einer Struktur-Datei

Feld	Name	Typ	L.nge	Dezimalstellen
1	Field_name	Zeichen	10	
2	Field_type	Zeichen	1	
3	Field_len	Numerisch	3	0
4	Field_dec	Numerisch	4	0

Mit COPY STRUCTURE EXTENDED k.nnen Sie in Ihren Applikationen die Struktur von Datenbanken auf einfache Art neu erzeugen oder ver.ndern. Um aus einer erweiterten Struktur-Datei eine neue Datenbank zu erzeugen, verwenden Sie CREATE FROM. Um eine leere Struktur-Datei zu erzeugen, verwenden Sie CREATE.

Hinweise

. Zeichenfelder, die l.nger als 255 Zeichen sind: Clipper unterst.tzt eine maximale L.nge von 64 KByte f.r Zeichenfelder. Aus Kompatibilit.tsgr.nden werden Feldl.ngen gr..er als 255 als eine Kombination von Field_len und Field_dec dargestellt. Nach Durchf.hrung des Befehles COPY STRUCTURE EXTENDED k.nnen Sie die L.nge eines Zeichenfeldes mit folgender Formel bestimmen:

```
nFeldLaenge = IF(Field_type = "C" .AND. Field_dec <> 0,;
   (Field_dec * 256) + Field_len, Field_len)
```

Beispiele

In diesem Beispiel wird die Datei Struk.DBF als erweiterte Struktur-Datei aus Verkauf.DBF erzeugt. Zur Verdeutlichung eines typischen Aufbaus von Felddefinitionen wird der Inhalt von Struk.DBF aufgelistet.

```
USE Verkauf NEW
COPY STRUCTURE EXTENDED TO Struk
USE Struk NEW
LIST Field_name, Field_type, Field_len, Field_dec
Ergebnis:
  1 Filiale     C 3   0
  2 Verkaeufer  C 4   0
  3 Kunde       C 4   0
  4 Produkt     C 25  0
  5 Umsatz      N 8   2
  6 Bemerkung   C 0 125   // Die Feldl.nge betr.gt 32.000 Zeichen
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

COPY TO Kopiert Datens.tze in eine neue .DBF- oder ASCII-Datei

---

Syntax

COPY [FIELDS <idFelderListe>] TO <xcDatei>
[<Wirkungsbereich>] [WHILE <lBedingung>] [FOR <lBedingung>]
[SDF | DELIMITED [WITH BLANK | <xcBegrenzer>]]

Argumente

FIELDS <idFelderListe> enth.lt eine Liste der Felder, die in die Zieldatei <xcDatei> kopiert werden sollen. Die Vorgabe umfa.t alle Felder.

TO <xcDatei> gibt den Namen der Zieldatei an. Der Name kann entweder als Literal oder als eingeklammerter Zeichenausdruck (erweiterter Ausdruck) angegeben werden. Ist kein Dateityp spezifiziert, wird .DBF als Vorgabe verwendet. Bei Verwendung von SDF oder DELIMITED wird .TXT als Vorgabetyp f.r die Quelldatei angenommen, au.er Sie geben eine andere Dateierweiterung an.

<Wirkungsbereich> spezifiziert den Teil, der von der Quell-Datenbank .bernommen und kopiert werden soll. Die Vorgabe umfa.t alle Datens.tze (ALL).

WHILE <lBedingung> spezifiziert die Gruppe der zu kopierenden Datens.tze, die sich vom aktuellen bis zum ersten, die Bedingung nicht erf.llenden Datensatz erstreckt.

FOR <lBedingung> selektiert alle Datens.tze innerhalb des <Wirkungsbereiches>, welche die angef.hrte Bedingung erf.llen und somit kopiert werden.

SDF erzeugt die Zieldatei als System Data Format ASCII File (also eine ASCII-Datei im Systemdatenformat). Jeder Datensatz besitzt eine feste, durch die Struktur der selektierten .DBF-Datei vorgegebene L.nge und ist durch ein Carriage Return/Line Feed begrenzt. Die Felder haben eine feste L.nge und sind durch kein Trennzeichen voneinander getrennt. Zeichenfelder werden mit nachfolgenden, numerische Felder mit f.hrenden Leerzeichen aufgef.llt. Datumsfelder werden im Format JJJJMMTT und logische Felder als T oder F dargestellt. Als Dateiende-Marke wird [Strg]+ [.] (1A hex oder CHR(26)) verwendet.

DELIMITED erzeugt die Zieldatei als ASCII-Textdatei. Datens.tze k.nnen in der L.nge variieren und werden durch Carriage Return/Line Feed voneinander getrennt. Felder sind ebenfalls variabel in ihrer L.nge und werden durch Kommata voneinander getrennt. Zeichenfelder werden in Anf.hrungszeichen (das sind die vorgegebenen Begrenzungszeichen) eingeschlossen. Die Begrenzungszeichen k.nnen durch den Zusatz WITH <xcBegrenzer> ge.ndert werden. F.hrende und nachfolgende Leerzeichen werden bei numerischen und Zeichenfeldern abgeschnitten. Datumsfelder werden im Format JJJJMMTT und logische Felder als T oder F dargestellt. Als Dateiende-Marke wird [Strg]+ [Z] (1A hex) verwendet.

DELIMITED WITH BLANK kennzeichnet eine ASCII-Textdatei, in der die Felder durch ein Leerzeichen voneinander getrennt und Zeichenfelder nicht in Begrenzern eingeschlossen sind.

DELIMITED WITH <xcBegrenzer> kennzeichnet eine ASCII-Textdatei, in der die Zeichenfelder durch das angegebene Begrenzungszeichen voneinander getrennt sind.
<xcBegrenzer> kann entweder als Literal oder als eingeklammerter Zeichenausdruck
angegeben werden.

Achtung

Wird die Klausel DELIMITED WITH in einer Befehlszeile mit COPY TO oder APPEND FROM angegeben, so mu. diese Klausel als letzte Klausel aufgef.hrt werden.

Anwendung

COPY TO ist ein Datenbankbefehl, der die selektierte Datenbank komplett oder teilweise in eine neue Datei kopiert. Die in der aktuellen Datenbank enthaltenen S.tze werden alle kopiert, soweit sie nicht durch einen der Zus.tze <Wirkungsbereich>, FOR oder WHILE oder einen Filter ausgeschlossen sind. Als gel.scht gekennzeichnete Datens.tze werden nicht .bernommen, wenn DELETED auf ON eingestellt ist. Ist ein FILTER aktiv, werden alle durch den Filter nicht sichtbaren S.tze ebenfalls nicht kopiert.

Die Datens.tze werden in der Sortierfolge des kontrollierenden Indexes kopiert, soweit eine Indexdatei im selektierten Arbeitsbereich ge.ffnet und SET ORDER nicht auf Null gesetzt worden ist. Anderenfalls werden die S.tze in der Reihenfolge ihres Auftretens kopiert.

Clipper .ffnet die Zieldatei in einer Netzwerkumgebung vor Ausf.hrung von COPY TO automatisch im Exclusive-Modus. Weitere Informationen zur Programmierung von Netzwerkanwendungen finden Sie im Handbuch Programmierung und Hilfsprogramme.

Beispiele

. Das folgende Beispiel demonstriert die Kopie einer kompletten Datenbank:

```
USE Verkauf NEW
COPY TO Temp
```

. Dieses Beispiel zeigt das Layout einer SDF-Datei mit vier Feldern, wobei f.r jeden Datentyp ein Feld vorhanden ist:

```
USE Test NEW
COPY NEXT 1 TO Temp SDF
TYPE Temp.TXT
```

Ergebnis-Datei:

```
Zeichen         12.0019891231T
```

. Dieses Beispiel zeigt das Aussehen einer DELIMITED-Datei:

```
COPY NEXT 1 TO Temp DELIMITED
TYPE Temp.TXT
```

Ergebnis-Datei:

```
"Zeichen",12.00,19891231,T
```

. Dieses Beispiel zeigt das Aussehen einer DELIMITED-Datei mit einem anderen Begrenzer:

```
COPY NEXT 1 TO Temp DELIMITED WITH '
TYPE Temp.TXT
```

Ergebnis-Datei:

```
'Zeichen',12.00,19891231,T
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

ADEL() L.scht Elemente eines Arrays

Syntax

ADEL(<aArray>, <nPosition>) -> aArray

Parameter

<aArray> ist das Array, in dem ein Element zu l.schen ist.

<nPosition> ist die Position des zu l.schenden Elements.

R.ckgabe

ADEL() liefert eine Referenz auf das Array <aArray>.

Anwendung

ADEL() ist eine Arrayfunktion zum L.schen einzelner Arrayelemente. Der Inhalt der angegebenen Elementposition wird gel.scht, und alle Elemente von dieser Position bis zum Ende des Arrays werden um ein Element aufger.ckt. Die letzte Position des Arrays erh.lt dann den Wert NIL. Dieser Wert bleibt solange erhalten, bis ihr ein neuer Wert zugewiesen wird.

Warnung

In Clipper sind mehrdimensionale Arrays als Verschachtelung mit anderen Arrays implementiert. Wird mit <aArray> ein mehrdimensionales Array angegeben, so l.scht ADEL() das mit <nPosition> angegebene Subarray, was zur Folge hat, da. <aArray> nicht mehr die urspr.ngliche Dimension besitzt. Grunds.tzlich sollte ADEL() nicht auf mehrdimensionale Arrays angewendet werden.

Beispiele

Dieses Beispiel erzeugt ein konstantes Array mit drei Elementen. Das zweite Element wird gel.scht und dadurch das dritte Element in die zweite Position ger.ckt. Ein Zugriff auf das dann undefinierte dritte Element ergibt den Wert NIL.

```
LOCAL aArray := { 1, 2, 3 }      // aArray = { 1, 2, 3 }
ADEL(aArray, 2)                  // aArray = { 1, 3, NIL }
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

COUNT Z.hlt die Datens.tze einer .DBF-Datei und weist das Ergebnis einer Variablen zu

---

Syntax

COUNT TO <idVar>
[<Wirkungsbereich>] [WHILE <lBedingung>] [FOR <lBedingung>]

Argumente

<idVar> benennt die Variable, der das Ergebnis von COUNT zugewiesen werden soll. Eine nicht existierende oder nicht sichtbare Variable wird als PRIVATE-Variable angelegt und ist somit nur in der laufenden Prozedur und in tieferen Ebenen liegenden Programmteilen verf.gbar.

<Wirkungsbereich> spezifiziert den Teil der Datens.tze in der aktuellen Datenbank, der
gez.hlt werden soll. Die Vorgabe umfa.t alle Datens.tze (ALL).

WHILE <lBedingung> spezifiziert die Gruppe der Datens.tze, die sich vom aktuellen bis zum ersten, die Bedingung nicht erf.llenden Datensatz erstreckt.

FOR <lBedingung> selektiert alle Datens.tze innerhalb des <Wirkungsbereiches>, welche die angef.hrte Bedingung erf.llen.

Anwendung

COUNT ermittelt die Anzahl der Datens.tze im aktuellen Arbeitsbereich, welche die angegebenen Bedingungen erf.llen und im <Wirkungsbereich> liegen. Das Ergebnis wird
<idVar> zugewiesen. <idVar> kann eine Variable jeder Speicherklasse, also auch ein Datenbankfeld (FIELD) sein.

Beispiele

. Das folgende Beispiel z.hlt alle Datens.tze in Verkauf.DBF, die mit der Filiale "100" zusammenh.ngen:

```
USE Verkauf NEW
COUNT TO nFilAnzahl FOR Filiale = "100"
? nFilAnzahl                              // Ergebnis: 4
```

. Dieses Beispiel zeigt, wie die ermittelte Anzahl an Datens.tzen in Verkauf.DBF dem Datenbankfeld Anzahl in Filiale.DBF zugewiesen wird:

```
USE Filiale INDEX Filiale NEW
SEEK 100
USE Verkauf INDEX VerkFil NEW
SEEK 100
COUNT TO Filiale->Anzahl WHILE Filiale = 100
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

CREATE Erzeugt eine leere .DBF-Datei, deren Struktur zur Definition von Datenbanken dient

---

Syntax

CREATE <xcExtendedDatenbank>

Argumente

<xcExtendedDatenbank> gibt den Namen der neuen, leeren Datenbank an, welche die Strukturdaten enthalten soll. Der Name kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Falls keine andere Erweiterung angegeben ist, wird die Standard-Erweiterung .DBF benutzt.

Anwendung

CREATE erzeugt eine leere Struktur-Datei mit folgendem Aufbau:

Tabelle 4-6: Struktur einer Struktur-Datei

Feld	Name	Typ	L.nge	Dezimalstellen
1	Field_name	Zeichen	10	
2	Field_type	Zeichen	1	
3	Field_len	Numerisch	3	0
4	Field_dec	Numerisch	4	0

Genau wie mit COPY STRUCTURE EXTENDED k.nnen Sie mit CREATE in Verbindung mit CREATE FROM neue Datenbanken erstellen. Anders als COPY STRUCTURE EXTENDED erzeugt CREATE eine leere Struktur-Datei, so da. eine andere Datenbank als Vorlage nicht vorhanden sein mu..

Nach Ausf.hrung von CREATE wird <xcExtendedDatenbank> automatisch im aktuellen Arbeitsbereich ge.ffnet.

Beispiele

Dieses Beispiel zeigt, wie eine Struktur-Datei erzeugt, die Definition eines Feldes vorgenommen und eine neue Datenbank aus der (um eine Felddefinition) erweiterten Struktur-Datei erstellt wird:

```
CREATE TempStru
APPEND BLANK
REPLACE Field_name WITH "Name";
  Field_type WITH "C";
  Field_len WITH 25;
  Field_dec WITH 0
CLOSE
CREATE NeuDatei FROM TempStru
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

CREATE FROM Erzeugt eine neue .DBF-Datei aus einer .DBF-Datei, deren Felder die neue Dateistruktur enthalten

---

Syntax

CREATE <xcDatenbank> FROM <xcExtendedDatenbank>

Argumente

<xcDatenbank> ist der Name der aus einer erweiterten Struktur-Datei neu zu erzeugenden Datenbank.

<xcExtendedDatenbank> ist der Name einer erweiterten Struktur-Datei, deren Felder den Aufbau der neu zu erzeugenden Datenbank enthalten.

Beide Argumente k.nnen als Literal oder eingeklammerte Zeichenausdr.cke angef.hrt werden. Falls keine andere Erweiterung angegeben ist, wird die Standard-Erweiterung .DBF benutzt.

Anwendung

CREATE FROM erzeugt eine neue Datenbank, wobei die Felddefinitionen den Datens.tzen einer erweiterten Struktur-Datei entnommen werden. Diese als Vorlage benutzte erweiterte Struktur-Datei mu. folgende vier Felder enthalten:

Tabelle 4-7: Struktur einer Struktur-Datei

Feld	Name	Typ	L.nge	Dezimalstellen
1	Field_name	Zeichen	10	
2	Field_type	Zeichen	1	
3	Field_len	Numerisch	3	0
4	Field_dec	Numerisch	4	0

Nach Ausf.hrung von CREATE FROM wird <xcDatenbank> automatisch im aktuellen Arbeitsbereich ge.ffnet.

Hinweise

. Data dictionaries: F.r Applikationen, die Data dictionaries verwalten, k.nnen Sie in die Struktur-Datei, zus.tzlich zu den o.g. vier Feldern, beliebige andere Felder mitaufnehmen. Damit k.nnen Sie z.B. die einzelnen Felder kommentieren oder auch Farb- und Formatinformationen, Bedingungen f.r VALID und WHEN u.a. angeben. Bei der Durchf.hrung von CREATE FROM verwendet Clipper nur die vier ben.tigten Datenbankfelder, alle anderen Felder werden dabei ignoriert. Die Reihenfolge dieser vier Felder im Datensatz-Aufbau ist egal.

. Zeichenfelder, die l.nger als 255 Zeichen sind: Clipper unterst.tzt zwei Methoden zur Definition von Zeichenfeldern, die gr..er als 255 sind:

Geben Sie die Feldl.nge als Kombination von Field_len und Field_dec an, wobei folgende Formeln zu beachten sind:

```
FIELD->Field_len := <nFeldl.nge> % 256
FIELD->Field_dec := INT(<nFeldl.nge>) / 256)
```

.ndern Sie in der Struktur-Datei die Feldgr..e f.r Field_len von 3 in 5 ab. Nun k.nnen Sie die Feldl.nge als einfache Zahl eingeben. Die Ver.nderung der Feldgr..e kann z.B. mit DBU.EXE erfolgen. Weitere Informationen zu den Datenbank-Utilities finden Sie im Handbuch Programmierung und Hilfsprogramme.

Beispiele

. Dieses Beispiel zeigt ein interaktives CREATE-Programm:

```
CreateDatabase("NeuDatei")
RETURN
FUNCTION CreateDatabase(cNeueDbf)
  CREATE TmpExt                // Erzeugt eine leere Struktur-Datei
  USE TmpExt
```

```
  lWeiter := .T.
  DO WHILE lWeiter             // Eingabe von neuen Felddefinitionen
    APPEND BLANK
    CLEAR
    @ 5, 0 SAY "Name.........:" GET Field_name
    @ 6, 0 SAY "Typ..........:" GET Field_type
    @ 7, 0 SAY "L.nge........:" GET Field_len
    @ 8, 0 SAY "Dezimalstellen:" GET Field_dec
    READ
    lWeiter := (.NOT. EMPTY(Field_name))
  ENDDO
  DELETE ALL FOR EMPTY(Field_name)  // L.scht alle leeren S.tze
  PACK
  CLOSE
  CREATE (cNeueDbf) FROM TmpExt      // Erzeugt neue .DBF-Datei
  ERASE TmpExt.DBF
RETURN NIL
```

. Das folgende Beispiel demonstriert die Definition von Zeichenfeldern,
die l.nger als 255 Zeichen lang sind, hier 4000 Zeichen:

```
APPEND BLANK
REPLACE Field_name WITH "Bemerkung";
  Field_type WITH "C";
  Field_len WITH 4000 % 256;
  Field_dec WITH INT(4000 / 256)
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

DELETE Markiert Datens.tze zur L.schung

Syntax

DELETE [<Wirkungsbereich>] [WHILE <lBedingung>] [FOR <lBedingung>]

Argumente

<Wirkungsbereich> spezifiziert den Teil, der in der aktuellen Datenbank gel.scht werden soll. Wurde kein <Wirkungsbereich> angegeben, so wird nur der jeweils aktuelle Datensatz gel.scht. Wurde eine Bedingung als Klausel angegeben, so umfa.t die Vorgabe alle Datens.tze (ALL).

WHILE <lBedingung> spezifiziert die Gruppe der Datens.tze, die sich vom aktuellen bis zum ersten, die Bedingung nicht erf.llenden Datensatz erstreckt.

FOR <lBedingung> selektiert alle Datens.tze innerhalb des <Wirkungsbereiches>, welche die angef.hrte Bedingung erf.llen und somit gel.scht werden.

Anwendung

DELETE ist ein Datenbankbefehl, der Datens.tze markiert, so da. sie mit SET DELETED ON herausgefiltert, mit DELETED() abgefragt oder mit PACK aus der Datenbank physisch entfernt werden k.nnen. Au.erdem werden gel.schte S.tze von Ausgabebefehlen wie LIST und DISPLAY mit dem Zeichen * gekennzeichnet. Die Markierung von gel.schten Datens.tzen kann mit RECALL zur.ckgesetzt werden. Wenn Sie alle S.tze einer Datenbank l.schen m.chten, so benutzen Sie besser den Befehl ZAP anstatt DELETE ALL und PACK.

In einer Netzwerkumgebung mu. ein zu l.schender, einzelner Datensatz zuerst mit RLOCK() gesperrt werden, bevor DELETE durchgef.hrt werden kann. Bei der gleichzeitigen L.schung von mehreren Datens.tzen mu. zuerst die komplette Datenbank mit FLOCK() oder USE EXCLUSIVE gesperrt worden sein. Weitere Informationen zur Programmierung von Netzwerkanwendungen finden Sie im Handbuch Programmierung und Hilfsprogramme.

Hinweise

. DELETE bei SET DELETED ON: Wenn Sie den aktuellen Datensatz l.schen, w.hrend DELETED auf ON eingestellt ist, bleibt dieser Satz bis zur ersten Ver.nderung des Datensatzzeigers sichtbar.

Beispiele

Das folgende Beispiel demonstriert den Einsatz der FOR-Bedingung, um mehrere S.tze zu l.schen:

```
USE Verkauf INDEX Verkauf NEW
DELETE ALL FOR BestDatum > CTOD("01.01.89")
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

DELETE FILE L.scht eine Datei von der Platte

Syntax

DELETE FILE | ERASE <xcDatei>

Argumente

<xcDatei> ist der Name der zu l.schenden Datei. Der Name kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Die Erweiterung des Dateinamens mu. mit angegeben werden. Falls erforderlich, k.nnen Sie auch das Laufwerk und/oder den Pfad angeben.

Anwendung

DELETE FILE ist ein Dateibefehl, der anstelle von ERASE zur L.schung einer Datei von der Platte verwendet werden kann. SET DEFAULT und SET PATH beeinflussen die Wirkungsweise von DELETE FILE nicht. Die zu l.schende Datei wird nur im aktuellen DOS-Pfad oder im bei <xcDatei> explizit angef.hrten Pfad gesucht.

Achtung

Dateien m.ssen vor der L.schung geschlossen werden.

Beispiele

Dieses Beispiel l.scht eine Datei von der Platte und .berpr.ft, ob der Vorgang erfolgreich durchgef.hrt worden ist:

```
? FILE("Temp.DBF")       // Ergebnis: .T.
DELETE FILE Temp.DBF
? FILE("Temp.DBF")       // Ergebnis: .F.
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

DIR* Gibt eine Liste der Dateien im angegebenen Pfad aus

Syntax

DIR [<xcDateiSpezifikation>]

Argumente

<xcDateiSpezifikation> spezifiziert die anzuzeigenden Dateien und kann Wildcards (die Zeichen * und ?) enthalten. <xcDateiSpezifikation> kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden.

Anwendung

DIR ist ein Dateibefehl, der eine Liste der Dateien im angegebenen Pfad in einem von zwei Formaten ausgibt. Das Ausgabeformat h.ngt von der Angabe des Zusatzes <xcDateiSpezifikation> ab. Ohne Zusatz wird eine Liste aller Datenbanken im aktuellen oder angegebenen Pfad angezeigt. Diese Standardliste enth.lt den Dateinamen, das Datum der letzten Aktualisierung und die Anzahl der Datens.tze von jeder Datenbank. Bei Angabe des Zusatzes <xcDateiSpezifikation> wird eine Liste der spezifizierten Dateien mit Dateinamen, Erweiterung, Anzahl der Bytes und Datum der letzten Aktualisierung ausgegeben.

Wurde keine Pfadangabe gemacht, zeigt DIR die Dateien des aktuellen DOS-Verzeichnisses an, au.er es wurde mit SET DEFAULT TO ein anderer Pfad voreingestellt.

DIR ist ein Kompatibilit.ts-Befehl und wird daher nicht f.r den Einsatz in neuen Applikationen empfohlen. Er wird durch die Funktion DIRECTORY() ersetzt, die ein Array mit Datei-Informationen entsprechend <xcDateiSpezifikation> zur.ckgibt. Auf der Basis dieses Arrays k.nnen Datei-Informationen angezeigt werden.

Hinweise

. Datei-Auswahlliste: Wenn der Benutzer eine Datei ausw.hlen soll, k.nnen Sie dies durch eine Kombination der Funktionen ADIR() und ACHOICE() erreichen. ADIR() gibt ein oder mehrere Arrays mit Datei-Informationen zur.ck, ACHOICE() erstellt aus einem Array ein Auswahlmen..

Beispiele

Die folgenden Beispiele zeigen die DOS- und Datenbank im aktuellen Verzeichnis an:

```
cDateiSpez = "*.*"
DIR (cDateiSpez)     // Listet alle Dateien auf
DIR                  // Listet alle .DBF-Dateien auf
DIR *.PRG            // Listet alle Programmdateien auf
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

DISPLAY Gibt Datens.tze auf der angegebenen Ausgabeeinheit aus

---

Syntax

DISPLAY <AusdrucksListe>
[TO PRINTER] [TO FILE <xcDatei>]
[<Wirkungsbereich>] [WHILE <lBedingung>] [FOR <lBedingung>] [OFF]

Argumente

<AusdrucksListe> gibt eine Liste der Ausdr.cke an, die f.r jeden verarbeiteten Datensatz angezeigt werden soll.

TO PRINTER f.hrt die Ausgabe zus.tzlich auf dem Drucker aus.

TO FILE leitet die Ausgabe zus.tzlich in die Datei <xcDatei> um. <xcDatei> kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Falls keine andere Erweiterung angegeben ist, wird die Standard-Erweiterung .TXT hinzugef.gt.

<Wirkungsbereich> spezifiziert den Teil der Datens.tze in der aktuellen Datenbank, der angezeigt werden soll. Wurde kein <Wirkungsbereich> angegeben, wird nur der jeweils aktuelle Datensatz angezeigt (= NEXT 1). Wurde eine Bedingung als Klausel angegeben, so umfa.t die Vorgabe alle Datens.tze (ALL).

WHILE <lBedingung> spezifiziert die Gruppe der Datens.tze, die sich vom aktuellen bis zum ersten, die Bedingung nicht erf.llenden Datensatz erstreckt.

FOR <lBedingung> selektiert alle Datens.tze innerhalb des <Wirkungsbereiches>, welche die angef.hrte Bedingung erf.llen und somit angezeigt werden.

OFF unterdr.ckt die Ausgabe der Satznummer.

Anwendung

DISPLAY ist ein Datenbankbefehl, der die Datens.tze des aktuellen Arbeitsbereiches sequentiell verarbeitet und sie als Ergebnisse von <AusdrucksListe> an die angegebene Ausgabeeinheit sendet. Die Ausgabe erfolgt in tabellarischer Form, wobei die Spalten durch ein Leerzeichen voneinander getrennt sind. Der DISPLAY-Befehl zeigt keine Spalten.berschriften an und stoppt nicht nach einer vorgegebenen Satzanzahl. DISPLAY ist mit LIST identisch, au.er da. der vorgegebene <Wirkungsbereich> NEXT 1 anstatt ALL ist.

Nach Aufruf von DISPLAY wird die Ausgabe an den Bildschirm und optional an den Drucker oder in eine Datei gesendet. Damit w.hrend der Ausgabe auf einen Drucker oder in eine Datei keine Bildschirmausgabe erfolgt, m.ssen Sie vor dem DISPLAY-Befehl SET CONSOLE auf OFF setzen.

Hinweise

. Unterbrechung der Ausgabe: Um dem Benutzer eine Unterbrechung der DISPLAY-Ausgabe zu erlauben, geben Sie bei der FOR-Bedingung eine Tastenabfrage und -.berpr.fung mit INKEY() an (vgl. Beispiel).

. Druckrand: DISPLAY ber.cksichtigt, wie jeder Konsolen-Befehl, die aktuelle Einstellung von SET MARGIN, wenn die Ausgabe auf dem Drucker erfolgt.

Beispiele

. Dieses Beispiel zeigt eine einfache Bildschirmausgabe und eine bedingungsabh.ngige Druckerausgabe:

```
USE Verkauf NEW
DISPLAY DATE(), TIME(), Filiale
DISPLAY Filiale, Verkaeufer FOR Umsatz > 500 TO PRINTER
```

. Das folgende Beispiel demonstriert den Einsatz von INKEY() zur

Unterbrechung von DISPLAY, um zu .berpr.fen, ob der Benutzer [Esc]
gedr.ckt hat:

```
#define K_ESC 27
USE Verkauf NEW
DISPLAY Filiale, Verkaeufer, Umsatz FOR INKEY() <> K_ESC
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

EJECT Setzt den Druckkopf auf den Anfang der n.chsten Seite

Syntax

EJECT

Anwendung

EJECT ist ein Druckbefehl, der einen Seitenvorschub (CHR(12)) an den Drucker sendet und PCOL() und PROW() auf Null zur.cksetzt. Wenn Sie eine Druckzeile ansprechen, die kleiner als die letzte Druckzeile nach Ausf.hrung von EJECT oder SETPRC() ist, so f.hrt Clipper automatisch einen EJECT-Befehl durch. Daher mu. die Abfolge des Ausdrucks in Ihren Applikationen logischerweise von links nach rechts und die Seite hinab geplant werden. Wenn Sie die internen Einstellungen f.r die Druckzeile und -spalte auf Null zur.cksetzen m.ssen, ohne einen Seitenvorschub zu senden, verwenden Sie SETPRC().

Beispiele

Dieses Beispiel druckt einen einfachen Bericht aus und verwendet EJECT, um bei Erreichen der maximalen Druckzeile einer Seite auf die n.chste Seite zu gelangen:

```
LOCAL nZeile := 99, nSeite := 0
USE Verkauf NEW
SET PRINTER ON
SET CONSOLE OFF
DO WHILE .NOT. EOF()
  IF nZeile > 55
    EJECT
    ? "Seite " + LTRIM(STR(++nSeite, 3))
    ? "Datum " + CTOD(DATE())
    ?
    ? "Verk.ufer", "Umsatz"
    ?
    nZeile := 6
  ENDIF
  ? Verkauf->Verkaeufer, Verkauf->Umsatz
  nZeile++
  SKIP
ENDDO
SET PRINTER OFF
SET CONSOLE ON
CLOSE
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

ERASE L.scht eine Datei von der Platte

Syntax

ERASE | DELETE FILE <xcDatei>

Argumente

<xcDatei> ist der Name der zu l.schenden Datei. Der Name kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Die Erweiterung des Dateinamens mu. angegeben werden. Falls erforderlich, k.nnen Sie auch das Laufwerk und/oder den Pfad angeben.

Anwendung

ERASE ist ein Dateibefehl, der anstelle von DELETE FILE zur L.schung einer Datei von der Platte verwendet werden kann. SET DEFAULT und SET PATH beeinflussen die Wirkungsweise von ERASE nicht. Die zu l.schende Datei wird nur im aktuellen DOS-Pfad oder im bei <xcDatei> explizit angef.hrten Pfad gesucht.

Achtung

Dateien m.ssen vor dem L.schen geschlossen werden.

Beispiele

Dieses Beispiel l.scht eine Datei von der Platte und .berpr.ft, ob der Vorgang erfolgreich durchgef.hrt worden ist:

```
? FILE("Temp.DBF")       // Ergebnis: .T.
DELETE FILE Temp.DBF
? FILE("Temp.DBF")       // Ergebnis: .F.
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

FIND* Durchsucht den Index nach einem angegebenen Schl.sselwert

---

Syntax

FIND <xcSuchString>

Argumente

<xcSuchString> ist der ganze Indexschl.ssel eines Datensatzes oder ein Teil des Schl.ssels, nach dem gesucht wird. <xcSuchString> kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. In letzterem Fall arbeitet FIND genauso wie SEEK.

Anwendung

FIND ist ein Datenbankbefehl, der einen Index nach dem ersten Schl.ssel durchsucht, der mit dem angegebenen Suchstring .bereinstimmt. Der Datensatzzeiger wird auf den gefundenen Datensatz gesetzt.
Abh.ngig von der Einstellung von SOFTSEEK ergeben sich nach einem FIND unterschiedliche Ergebnisse. Ist SOFTSEEK auf OFF gestellt, wird der Datensatzzeiger bei erfolgloser Suche auf LASTREC()+1 gesetzt, EOF() gibt true (.T.) und FOUND() false (.F.) zur.ck. Ist SOFTSEEK auf ON gestellt, wird der Datensatzzeiger bei erfolgloser Suche auf den ersten Satz positioniert, dessen Schl.sselwert gr..er als das Suchargument ist. FOUND() gibt false (.F.) zur.ck. EOF() gibt in diesem Fall nur dann true (.T.) zur.ck, wenn es im Index keine Schl.sselwerte gibt, die gr..er als das Suchargument sind.

FIND ist ein Kompatibilit.ts-Befehl und wird daher nicht f.r den Einsatz in neuen Applikationen empfohlen. Er wird durch den SEEK-Befehl ersetzt.

Beispiele

Diese Beispiele zeigen m.gliche Ergebnisse von FIND:

```
USE Verkauf INDEX Verkauf NEW
FIND ("500")
? FOUND(), EOF(), RECNO()   // Ergebnis: .F. .T. 85
FIND 200
? FOUND(), EOF(), RECNO()   // Ergebnis: .T. .F. 5
FIND "100"
? FOUND(), EOF(), RECNO()   // Ergebnis: .T. .F. 1
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

GO Setzt den Datensatzzeiger auf einen bestimmten Datensatz

Syntax

GO [TO] <nDatensatz> | BOTTOM | TOP

Argumente

<nDatensatz> gibt die Nummer des Zieldatensatzes an.

BOTTOM gibt den letzten logischen Datensatz im aktuellen Arbeitsbereich an.

TOP gibt den ersten logischen Datensatz im aktuellen Arbeitsbereich an.

Anwendung

GO ist ein Datenbankbefehl, der den Datensatzzeiger im aktuellen Arbeitsbereich auf den angegebenen Datensatz positioniert. Dieser Datensatz kann dabei als Zahl, Anfang oder Ende der Datei angegeben werden. Wird GO mit dem Zusatz TOP aufgerufen, wird der Datensatzzeiger im Falle eines aktiven Indexes auf den ersten Datensatz im Index bewegt, anderenfalls auf den Datensatz 1. Wird GO mit dem Zusatz BOTTOM aufgerufen, so wird der Datensatzzeiger im Falle eines aktiven Indexes auf den letzten Datensatz im Index bewegt, anderenfalls auf LASTREC().

Die Aufrufform GO <nDatensatz> entspricht der RECORD-Klausel jedes anderen Datenbankbefehles, der einen Wirkungsbereich unterst.tzt.

Hinweise

. Zugriffe auf gefilterte Datens.tze: Auf Datens.tze, die durch SET DELETED ON oder SET FILTER TO herausgefiltert worden sind, kann mit GO <nDatensatz> zugegriffen werden. Ist DELETED auf ON eingestellt und/oder ein aktiver Filter vorhanden, setzt GO BOTTOM den Datensatzzeiger auf den letzten logischen Datensatz, der nicht gel.scht ist und/oder die Filterbedingung erf.llt. Ist DELETED auf ON eingestellt und/oder ein aktiver Filter vorhanden, setzt GO TOP den Datensatzzeiger auf den ersten logischen Datensatz, der nicht gel.scht ist und/oder die Filterbedingung erf.llt.

. UNIQUE-Indizes: Auf Datens.tze, die nicht in einem mit SET UNIQUE ON oder INDEX...UNIQUE erstellten Index aufgenommen worden sind, kann mit GO <nDatensatz> zugegriffen werden.

. Nicht vorhandene Datensatznummern: Wenn Sie als Argument einen <nDatensatz> angeben, dessen Satznummer gr..er als LASTREC() ist, erzeugt Clipper keinen Laufzeitfehler. Stattdessen wird der Datensatzzeiger auf LASTREC()+1 bewegt und EOF() sowie BOF() werden auf true (.T.) gesetzt.

. Aktualisierung von Datenbank- und Indexpuffern: In einer Netzwerkumgebung k.nnen Sie die Datenbank- und Indexpuffer aktualisieren, ohne den Datensatzzeiger mit GO RECNO() zu bewegen. Weitere Informationen zur Programmierung von Netzwerkanwendungen finden Sie im Handbuch Programmierung und Hilfsprogramme.

Beispiele

Diese Beispiele zeigen einfache Ergebnisse von GO:

```
USE Verkauf NEW
? LASTREC()     // Ergebnis: 84
//
GO TOP
? RECNO()       // Ergebnis: 1
//
GO BOTTOM
? RECNO()       // Ergebnis: 84
//
GO 5
? RECNO()       // Ergebnis: 5
```

```
//
GO 5 + 15
? RECNO()        // Ergebnis: 20
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

INDEX Erzeugt eine Indexdatei

Syntax

INDEX ON <Schl.sselAusdruck> TO <xcIndexdatei> [UNIQUE]

Argumente

<Schl.sselAusdruck> ist ein Ausdruck, der den Schl.sselwert f.r jeden Datensatz im aktuellen Arbeitsbereich zur Eintragung in den Index zur.ckgibt. <Schl.sselAusdruck> kann ein Zeichen-, Datums-, numerischer oder logischer Typ sein und bis zu 250 Zeichen lang sein.

TO <xcIndexdatei> gibt den Namen der zu erzeugenden Indexdatei an. Der Name kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Falls keine andere Erweiterung angegeben ist, wird eine Standard-Erweiterung benutzt. F.r Standard-Indexdateien von Clipper wird die Erweiterung .NTX und f.r dBASE III PLUS- kompatible Indizes die Erweiterung .NDX benutzt.

UNIQUE bewirkt, da. nur der jeweils erste von mehreren gleichen Schl.sselwerten in die <xcIndexdatei> aufgenommen wird.

Anwendung

INDEX ist ein Datenbankbefehl, der eine Datei erzeugt, die einen Index von Datens.tzen der aktuellen Datenbank enth.lt. Der Index basiert auf dem <Schl.sselAusdruck>. Wenn Sie die <xcIndexdatei> benutzen, erscheinen die Datens.tze der aktuellen Datenbank in der nach dem <Schl.sselAusdruck> sortierten Reihenfolge. Die nat.rliche, nach Satznummern geordnete, auf der Platte physisch vorhandene Reihenfolge der Datenbank wird allerdings nicht ver.ndert. INDEX sortiert Schl.sselausdr.cke vom Typ Zeichen entsprechend den ASCII-Werten jedes einzelnen Zeichens im Ausdruck. Numerische Ausdr.cke werden entsprechend ihres Wertes und Datumsausdr.cke in chronologischer Folge geordnet, wobei ein leeres Datum immer am Indexanfang eingeordnet wird. Bei logischen Werten wird false (.F.) am Indexanfang und true (.T.) am Indexende eingeordnet. Memo-Felder k.nnen nicht indiziert werden.

Nach Aufruf von INDEX ON werden alle ge.ffneten Indexdateien im aktuellen Arbeitsbereich geschlossen und die neue Indexdatei erzeugt. Nach dieser Operation wird der neue Index aktiviert und der Datensatzzeiger auf den ersten Satz im Index bewegt.

INDEX .hnelt SORT, wobei bei INDEX jedoch keine physische Kopie der Datenbank erstellt und der Index bei Eingabe eines neuen Schl.sselwertes jeweils aktualisiert wird. SORT dient prim.r zur Anfertigung einer sortierten Kopie der aktuellen Datenbank oder eines Teils davon.

INDEX .ffnet die <xcIndexdatei> in einer Netzwerkumgebung exclusive. Falls dies fehlschl.gt, tritt ein Laufzeitfehler auf. Weitere Informationen zur Programmierung von Netzwerkanwendungen finden Sie im Handbuch Programmierung und Hilfsprogramme.

Hinweise

. Index auf ein Datumsfeld: Clipper unterst.tzt die Indizierung von Datumsfeldern sowohl f.r .NTX- als auch f.r .NDX-Indextypen. Wird ein Datum als Teil eines Schl.sselausdruckes ben.tigt, m.ssen Sie den gesamten <Schl.sselAusdruck> in eine Zeichenkette umwandeln, wobei das Datum mit DTOS() bearbeitet werden mu..

. Deklarierte Variablen: LOCAL- und STATIC-Variablen k.nnen nicht in einem Indexschl.ssel verwendet werden. Der Grund hierf.r ist, da. Indexausdr.cke in der Indexdatei als Text abgelegt und sp.ter .ber den Makro-Operator & compiliert, ausgef.hrt und in die tats.chlichen Schl.sselwerte umgewandelt werden. Da aber LOCAL- und STATIC-Variablen f.r Makro-Variablen und -Ausdr.cke nicht sichtbar sind, k.nnten sie in Index-Schl.sselausdr.cken auch nicht korrekt aufgel.st werden. Aus .hnlichen Gr.nden ist eine Variable, die .ber eine der anderen Deklarationen zur Compilierzeit (z.B. MEMVAR und FIELD) erzeugt worden

ist, f.r Index-Schl.sselausdr.cke nicht zul.ssig. M.ssen Sie Variablen in einem Index-Schl.sselausdruck .ber eine Alias- oder andere Referenz ansprechen, sollten Sie dies explizit im Ausdruck und nicht .ber Compiler-Deklarationen vornehmen. Weitere Informationen zu Variablen finden Sie in diesem Handbuch im Kapitel Grundkonzepte.

. Gel.schte und gefilterte Datens.tze: Zur L.schung markierte oder gefilterte Datens.tze werden in den Index mit aufgenommen.

. Indizes in absteigender Folge: Zur Erzeugung von Indizes mit absteigender Sortierfolge verwenden Sie DESCEND(). Diese Funktion verarbeitet jeden Datentyp und gibt das Komplement des jeweiligen Wertes zur.ck. Bei einer Suche mit SEEK verwenden Sie dann DESCEND() als Teil des Suchargumentes.

. Deutsche Duden-Sortierung: Damit Zeichenwerte entsprechend dem Duden sortiert werden, geben Sie UPPER(<Schl.sselAusdruck>) als Argument an. Bei der Sortierung nach Duden wird die Gro.- und Kleinschreibung ignoriert.

. Kompatible Indexdateien: Clipper unterst.tzt dBASE III PLUS-kompatible Indexdateien .ber einen speziellen Datenbank-Treiber. Indexschl.ssel sind bei .NDX-Dateien auf 100 Zeichen L.nge beschr.nkt. Weitere Informationen zum Linken von Treibern finden Sie im Handbuch Programmierung und Hilfsprogramme im Kapitel Linker - RTLink.

Achtung
Der dBASE III PLUS-Datenbank-Treiber unterst.tzt den Sperrmechanismus von Clipper, nicht aber den von dBASE III PLUS. Daher k.nnen Sie in einer Netzwerkumgebung nicht gleichzeitig mit einer Clipper-Applikation und einem dBASE III PLUS-Programm auf ein und dieselbe Indexdatei zugreifen, ohne die Datensicherheit zu gef.hrden. Dies ist nur mit zwei oder mehreren Clipper-Programmen m.glich, da der Sperrmechanismus der Datenbank-Treiber f.r .NTX- und .NDX-Indexdateien identisch ist.

. UNIQUE-Indizes: Eine Indizierung mit INDEX...UNIQUE oder der Einstellung SET UNIQUE ON bewirkt, da. Clipper einen Index erstellt, in den von mehreren gleichen Schl.sselwerten nur der jeweils erste Wert aufgenommen wird. Dieses Verhalten beh.lt der so erzeugte Index f.r alle Operationen bei, wie z.B. bei einer Index-Aktualisierung und -Reindizierung oder bei Ausf.hrung des PACK-Befehles. Auch sp.teres Ein- und Ausschalten von SET UNIQUE ver.ndert das Verhalten dieses Indexes nicht mehr.

. TRIM() von Schl.sselwerten: Schl.sselausdr.cke, welche die L.nge des Schl.ssels wechseln, k.nnen in Clipper einen nicht verarbeitbaren Index erzeugen. Dies liegt darin begr.ndet, da. Clipper die L.nge des Index-Schl.sselausdruckes zuerst anhand eines leeren Datensatzes berechnet und alle Schl.sselausdr.cke entsprechend dieser L.nge in festen Abst.nden abspeichert. Ein mit TRIM() abgeschnittener <Schl.sselAusdruck> ergibt daher immer einen Nullstring, so da. eine Differenz in den L.ngen des aktuellen und zuerst berechneten Schl.sselwertes vorliegt. Daher mu. ein mit TRIM() behandelter <Schl.sselAusdruck> immer mit PADR() auf eine feste L.nge aufgef.llt werden, z.B.:

```
USE Kunden NEW
INDEX ON PADR(RTRIM(Nachname) + Vorname, 40) TO Kunden
```

Hinweis
Verwenden Sie nach M.glichkeit unterschiedliche Namen f.r eine Indexdatei und ein Feld.

Beispiele
. Dieses Beispiel zeigt einige einfache Ergebnisse von INDEX:

```
? TYPE("Filiale")            // Ergebnis: C
INDEX ON Filiale TO Laden
```

```
? TYPE("Umsatz")              // Ergebnis: N
INDEX ON Umsatz TO Verkauf
? TYPE("Datum")               // Ergebnis: D
INDEX ON Datum TO Aktuell
INDEX ON Filiale + STR(Umsatz) + DTOS(Datum) TO Alles
```

. Die folgenden Beispiele demonstrieren die Erstellung von absteigenden Indizes:

```
USE Rechnung NEW
INDEX ON DESCEND(RgDatum) TO RgDatum
INDEX ON Kunde + DESCEND(DTOS(RgDatum)) TO RgDatum
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

ADIR()* F.llt ein Array mit Informationen des DOS-Verzeichnisses

---

Syntax

```
ADIR([<cDateimaske>],
    [<aName>],
    [<aGr..e>],
    [<aDatum>],
    [<aZeit>],
    [<aAttribute>]) -> nAnzahlDateien
```

Parameter

<cDateimaske> ist eine Dateimaske, mit der im DOS-Verzeichnis nach Dateien gesucht werden soll. Die Dateimaske entspricht der Standard-Dateispezifikation und kann sowohl die Platzhalter * und ?, als auch eine vorangestellte Laufwerks- und Verzeichnisangabe enthalten. Wird der Parameter nicht angegeben, so wird *.* angenommen.

<aName> ist das Array, dessen Elemente mit den gefundenen Dateinamen gef.llt werden. Jedes Element enth.lt dann den Dateinamen zusammen mit der Dateinamenserweiterung in Gro.buchstaben.

<aGr..e> ist das Array, das mit den Dateigr..en der jeweiligen Dateien in <aName> gef.llt wird. Jedes Element ist dann vom Typ numerisch.

<aDatum> ist das Array, das mit den Datumsangaben der Dateien in <aName> gef.llt wird. Jedes Element ist dann vom Typ Datum.

<aZeit> ist das Array, das mit den Speicherzeiten der Dateien in <aName> gef.llt wird. Jedes Element ist dann vom Typ Zeichen in der Form hh:mm:ss.

<aAttribute> ist das Array, das mit den Dateiattributen der Dateien in <aName> gef.llt wird. Jedes Element enth.lt den Typ Zeichen. Sobald Sie <aAttribute> angegeben haben, werden alle Dateien gefunden, egal ob ihre Attribute Hidden, System oder Directory sind. Wird <aAttribute> nicht angegeben, werden nur die normalen Dateien gefunden.

R.ckgabe

ADIR() liefert als R.ckgabewert die Anzahl der gefundenen Dateien als numerischen Wert.

Anwendung

ADIR() ist eine Arrayfunktion, die zwei wichtige Operationen durchf.hrt: zum einen liefert sie die Anzahl der .ber die Dateimaske gefundenen Dateien, zum zweiten f.llt sie die als Parameter angegebenen Arrayvariablen mit Dateinamen, Dateigr..en, Dateidatum, Dateizeiten und Dateiattributen.

ADIR() wurde nur aus Kompatibilit.tsgr.nden implementiert und ist daher f.r neue Applikationen nicht mehr zu empfehlen. Eine vergleichbare Funktion ist DIRECTORY(), welche keine vordefinierten Arrays erwartet und alle Dateiinformationen in einem einzigen mehrdimensionalen Array als Ergebnis liefert.

Hinweise

. Platzhalter: Der Parameter <cDateimaske>, der einen Dateinamen spezifiziert, kann die Platzhalter * und ? enthalten. Das Zeichen * steht f.r eine beliebige Gruppe von Zeichen, ? steht f.r ein beliebiges Zeichen. Zum Beispiel: D* trifft zu auf DRIN, DRAUSSEN, DRAN, DABEI, DRANG, wohingegen DR?N lediglich auf DRIN und DRAN zutrifft.

. Verzeichnis: Wird der Parameter <aAttribute> angegeben und ist die Dateimaske <cDateimaske> als *.* spezifiziert, so werden in <aNamen> auch Verzeichnisse abgelegt. Die korrespondierenden Eintr.ge in <aAttribute> enthalten die Kennung "D" f.r Directory. Wird ADIR() in einem Unterverzeichnis ausgef.hrt, so sind die ersten beiden Eintr.ge in <aNamen> "." und "..", die Zeichen f.r aktuelles und Ausgangs-Verzeichnis. In allen F.llen enthalten <aDatum> und <aZeit> das Datum und die Uhrzeit der Erstellung der Verzeichnisse. <aGr..e> wird dabei immer mit 0

angegeben.

Beispiele

Dieses Beispiel erzeugt ein Array, das die Namen aller Dateien enth.lt, die auf ".TXT" enden:

```
LOCAL aDateien[ADIR("*.TXT")]
ADIR("*.TXT",aDateien)
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

INPUT Weist das Ergebnis eines eingegebenen Ausdrucks einer Variablen zu

---

Syntax

INPUT [<PromptAusdruck>] TO <idVar>

Argumente

<PromptAusdruck> gibt einen Ausdruck beliebigen Datentyps an, der optional vor dem Eingabefeld angezeigt wird.

<idVar> gibt die Variable an, der die Tastatureingaben zugewiesen werden sollen. Wenn diese Variable nicht existiert oder nicht sichtbar ist, wird eine PRIVATE-Variable angelegt und dieser das Eingaberesultat zugewiesen.

Anwendung

INPUT ist ein Konsolen-Befehl und Wartezustand, der Eingaben von der Tastatur entgegennimmt, die einen g.ltigen Ausdruck beliebigen Datentyps mit bis zu 255 Zeichen L.nge ergeben. Nach dem Aufruf von INPUT werden ein Carriage Return/Line Feed und der <PromptAusdruck> auf dem Bildschirm ausgegeben. Unmittelbar nach dem <PromptAusdruck> beginnt der Eingabebereich. Eingaben, die .ber MAXCOL() hinausgehen, werden in der n.chsten Zeile fortgesetzt.

Die Taste [Return] beendet die Eingabe. Der eingegebene Ausdruck wird danach .ber den Makro-Operator & compiliert und ausgewertet, das Ergebnis wird <idVar> zugewiesen. Wurde keine Eingabe gemacht, wird NIL zugewiesen. Wenn <idVar> nicht existiert oder nicht sichtbar ist, wird sie als PRIVATE-Variable angelegt.

INPUT unterst.tzt nur zwei spezielle Tasten, ### und [Return], nicht jedoch [Esc]. ### l.scht die jeweils letzte Eingabe, w.hrend [Return] den Eingabeausdruck .bernimmt und den Eingabemodus als einzige Taste beendet.

Beispiele

Dieses Beispiel zeigt, wie einer existierenden LOCAL-Variablen .ber INPUT ein Ausdruck zugewiesen werden kann:

```
LOCAL Ausdruck
INPUT "Ausdruck: " TO Ausdruck
IF Ausdruck <> NIL
  ? Ausdruck
ELSE
  ? "Kein Ausdruck eingegeben"
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

JOIN Erzeugt eine neue Datenbank durch Verschmelzung der Datens.tze und -felder von zwei Arbeitsbereichen

---

Syntax

JOIN WITH <xcAlias> TO <xcDatenbank>
FOR <lBedingung> [FIELDS <idFelderListe>]

Argumente

WITH <xcAlias gibt den Namen des anderen Arbeitsbereiches an, der mit den Datens.tzen der aktuellen Datenbank verschmolzen werden soll. Der Name kann als Literal oder eingeklammerter Zeichenausdruck angegeben werden.

<xcDatenbank> ist der Name der Zieldatei. Der Name kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden.

FOR <lBedingung> selektiert alle Datens.tze, welche die angef.hrte Bedingung erf.llen.

FIELDS <idFelderListe> gibt die Felder an, die in die neue Datenbank .bernommen werden sollen. Datenbankfelder des anderen Arbeitsbereichs <xcAlias> werden durch die Angabe des Alias-Namens gekennzeichnet. Wurde keine FIELDS-Klausel angegeben, werden alle Felder des aktuellen Arbeitsbereiches in die Zieldatei .bernommen.

Anwendung

JOIN erzeugt eine neue Datenbank durch Verschmelzung der Datens.tze und -felder von zwei Arbeitsbereichen anhand einer generellen Bedingung. JOIN funktioniert, indem es f.r jeden Datensatz im aktuellen, prim.ren Arbeitsbereich einen kompletten Durchlauf aller Datens.tze im sekund.ren Arbeitsbereich durchf.hrt, wobei diese auf die Erf.llung von <lBedingung> hin .berpr.ft werden. Ergibt <lBedingung> true (.T.), wird ein neuer Datensatz an die Zieldatei angehangen, der die angegebenen Felder beider Arbeitsbereiche enth.lt.

Achtung

Die Anzahl der verarbeiteten Datens.tze errechnet sich wie folgt: LASTREC() des prim.ren Arbeitsbereiches multipliziert mit LASTREC() des sekund.ren Arbeitsbereiches. Bei zwei Datenbanken mit jeweils 100 S.tzen w.rden also 10.000 Zugriffe mit Verarbeitung erfolgen. Daher sollte man vorher gut abw.gen, ob der Einsatz dieses Befehles sinnvoll bzw. notwendig ist.

Beispiele

Das folgende Beispiel demonstriert den Einsatz von JOIN, um aus den Dateien Kunden.DBF und Rechnung.DBF die Datei Kauf.DBF zu produzieren:

```
USE Rechnung NEW
USE Kunden NEW
JOIN WITH Rechnung TO Kauf;
  FOR Zuletzt = Rechnung->Zuletzt;
  FIELDS Zuerst, Zuletzt, Rechnung->Nummer, Rechnung->Umsatz
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

KEYBOARD F.llt den Tastatur-Eingabepuffer mit einem String

Syntax

KEYBOARD <cString>

Argumente

<cString> ist die in den Tastatur-Eingabepuffer abzulegende Zeichenkette.

Anwendung

KEYBOARD ist ein Tastaturbefehl, der den Tastatur-Eingabepuffer mit <cString> f.llt, nachdem alle ausstehenden Tasteneingaben aus dem Puffer gel.scht worden sind. Die Zeichen verbleiben im Puffer, bis sie durch einen Wartezustand wie ACCEPT, INPUT, READ, ACHOICE(), MEMOEDIT(), DBEDIT() oder INKEY() ausgelesen werden. <cString> kann zus.tzlich zu am Bildschirm darstellbaren Zeichen auch Steuerzeichen enthalten. Zur Umwandlung eines INKEY()-Codes in ein Zeichen wird normalerweise die Funktion CHR() benutzt, wobei aber nur INKEY()-Codes von Null bis 255 in den Tastatur-Eingabepuffer abgelegt werden k.nnen.

Beispiele

In diesem Beispiel wird KEYBOARD benutzt, um die R.ckgabe einer Auswahlliste in ein GET-Feld zu schreiben:

```
#include "Inkey.CH"
LOCAL cVar1 := SPACE(10), nVar := 2500, cVar2 := SPACE(10)
CLEAR
@ 09, 10 GET cVar1
@ 10, 10 GET cVar2 WHEN AuswahlListe()
@ 11, 10 GET nVar
READ
RETURN
FUNCTION AuswahlListe
  STATIC aListe := { "Eins", "Zwei", "Drei", "Vier" }
  LOCAL cBildschirm, nWahl, nTaste := LASTKEY()
  cBildschirm := SAVESCREEN(10, 10, 14, 20)
  @ 10, 10 TO 14, 20 DOUBLE
  IF (nWahl := ACHOICE(11, 11, 13, 19, aListe)) <> 0
    KEYBOARD CHR(K_CTL_Y) + aList[nWahl] + CHR(nTaste)
  ENDIF
  RESTSCREEN(10, 10, 14, 20, cBildschirm)
RETURN .T.
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist Inkey.CH

See Also:

LABEL FORM Gibt Labels auf der ausgew.hlten Ausgabeeinheit aus

---

Syntax

LABEL FORM <xcLabelDatei>
[TO PRINTER] [TO FILE <xcDatei>] [<Wirkungsbereich>]
[WHILE <lBedingung>] [FOR <lBedingung>] [SAMPLE]

Argumente

<xcLabelDatei> ist der Name der .LBL-Datei, die das Layout des Labels beinhalten soll. Der Name kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Falls keine andere Erweiterung angegeben ist, wird die Standard-Erweiterung .LBL benutzt.

TO PRINTER f.hrt die Ausgabe zus.tzlich auf dem Drucker aus.

TO FILE leitet die Ausgabe zus.tzlich in die Datei <xcDatei> um. <xcDatei> kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Falls keine andere Erweiterung angegeben ist, wird die Standard-Erweiterung .TXT hinzugef.gt.

<Wirkungsbereich> spezifiziert den Teil der Datens.tze in der aktuellen Datenbank, der in Form von Labels ausgegeben werden soll. Die Vorgabe umfa.t alle Datens.tze (ALL).

WHILE <lBedingung> spezifiziert die Gruppe der Datens.tze, die sich vom aktuellen bis zum ersten, die Bedingung nicht erf.llenden Datensatz erstreckt.

FOR <lBedingung> selektiert alle Datens.tze innerhalb des <Wirkungsbereiches>, welche die angef.hrte Bedingung erf.llen und somit angezeigt werden.

SAMPLE gibt ein Test-Label aus, wobei die jeweiligen Zeilen durch Sternchen dargestellt werden. Nach jeder Ausgabe erfolgt eine Abfrage, ob Sie weitere Test-Labels ausgeben m.chten. Wenn Sie dies bejahen, wiederholt sich die Test-Ausgabe, anderenfalls werden die Labels mit den tats.chlichen Daten ausgegeben.

Anwendung

LABEL FORM ist ein Konsolen-Befehl, der die Datens.tze im aktuellen Arbeitsbereich sequentiell verarbeitet und sie als Labels entsprechend des in <xcLabelDatei> abgelegten Layouts ausgibt. M.glichkeiten zur Erzeugung einer <xcLabelDatei> sind z.B. durch das Hilfsprogramm RL.EXE gegeben. Weitere Informationen zur Erstellung von Label-Layouts finden Sie im Handbuch Programmierung und Hilfsprogramme im Kapitel Reports und Labels.

Nach dem Aufruf von LABEL FORM wird die Ausgabe an den Bildschirm und optional an den Drucker oder in eine Datei gesendet. Damit w.hrend der Ausgabe auf einen Drucker oder in eine Datei keine Bildschirmausgabe erfolgt, m.ssen Sie vorher SET CONSOLE auf OFF setzen.

LABEL FORM sucht die <xcLabelDatei> zuerst im aktuellen Verzeichnis, soweit kein anderer Pfad mit angegeben worden ist. Wurde die Datei nicht gefunden, wird im mit SET PATH angegebenen Pfad gesucht.

Hinweise

. Unterbrechung der Ausgabe: Um dem Benutzer eine Unterbrechung der LABEL FORM-Ausgabe zu erlauben, geben Sie bei der FOR-Bedingung eine Tastenabfrage und -.berpr.fung mit INKEY() an (vgl. Beispiel).

. Druckrand: LABEL FORM ber.cksichtigt, wie jeder Konsolen-Befehl, die aktuelle Einstellung von SET MARGIN, wenn die Ausgabe auf dem Drucker erfolgt.

Beispiele

. Dieses Beispiel druckt mit einem einzigen Befehl mehrere Labels und

schreibt sie gleichzeitig in eine Datei. Der Befehl wird in zwei Varianten gezeigt:

```
LOCAL cLabel := "Verkauf", cDatei := "Verkauf"
USE Verkauf INDEX Verkauf NEW
LABEL FORM Verkauf TO PRINTER TO FILE Verkauf
LABEL FORM (cLabel) TO PRINTER TO FILE (cDatei)
```

. Das folgende Beispiel demonstriert den Einsatz von INKEY() zur Unterbrechung von LABEL FORM, um zu .berpr.fen, ob der Benutzer die Taste [Esc] gedr.ckt hat:

```
#define K_ESC 27
USE Verkauf INDEX Verkauf NEW
LABEL FORM Verkauf FOR INKEY() <> K_ESC
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

LIST Gibt Datens.tze auf der ausgew.hlten Ausgabeeinheit aus

Syntax

LIST <AusdrucksListe>
[TO PRINTER] [TO FILE <xcDatei>]
[<Wirkungsbereich>] [WHILE <lBedingung>] [FOR <lBedingung>] [OFF]

Argumente

<AusdrucksListe> ist eine Liste der f.r jeden verarbeiteten Datensatz auszuwertenden und auszugebenden Ausdr.cke.

TO PRINTER f.hrt die Ausgabe zus.tzlich auf dem Drucker aus.

TO FILE leitet die Ausgabe zus.tzlich in die Datei <xcDatei> um. <xcDatei> kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Falls keine andere Erweiterung angegeben ist, wird die Standard-Erweiterung .TXT hinzugef.gt.

<Wirkungsbereich> spezifiziert den Teil der Datens.tze in der aktuellen Datenbank, der ausgegeben werden soll. Die Vorgabe umfa.t alle Datens.tze (ALL).

WHILE <lBedingung> spezifiziert die Gruppe der Datens.tze, die sich vom aktuellen bis zum ersten, die Bedingung nicht erf.llenden Datensatz erstreckt.

FOR <lBedingung> selektiert alle Datens.tze innerhalb des <Wirkungsbereiches>, welche die angef.hrte Bedingung erf.llen und somit angezeigt werden.

OFF unterdr.ckt die Ausgabe der Satznummer.

Anwendung

LIST ist ein Datenbank-Befehl, der die Datens.tze des aktuellen Arbeitsbereiches sequentiell verarbeitet und sie als Ergebnisse von <AusdrucksListe> an die angegebene Ausgabeeinheit sendet. Die Ausgabe erfolgt in tabellarischer Form, wobei die Spalten durch ein Leerzeichen voneinander getrennt sind. LIST ist mit DISPLAY identisch, au.er da. der vorgegebene <Wirkungsbereich> ALL anstatt NEXT 1 ist.

Nach Aufruf von LIST wird die Ausgabe an den Bildschirm und optional an den Drucker oder in eine Datei gesendet. Damit w.hrend der Ausgabe auf einen Drucker oder in eine Datei keine Bildschirmausgabe erfolgt, m.ssen Sie vor dem LIST-Befehl SET CONSOLE auf OFF setzen.

Hinweise

. Unterbrechung der Ausgabe: Um dem Benutzer eine Unterbrechung der LIST-Ausgabe zu erlauben, geben Sie bei der FOR-Bedingung eine Tastenabfrage und -.berpr.fung mit INKEY() an (vgl. Beispiel).

. Druckrand: LIST ber.cksichtigt wie jeder Konsolen-Befehl die aktuelle Einstellung von SET MARGIN, wenn die Ausgabe auf dem Drucker erfolgt.

Beispiele

. Dieses Beispiel zeigt eine einfache Bildschirmausgabe und eine bedingungsabh.ngige Druckerausgabe:

```
USE Verkauf NEW
LIST DATE(), TIME(), Filiale
LIST Filiale, Verkaeufer FOR Umsatz > 500 TO PRINTER
```

. Das folgende Beispiel demonstriert den Einsatz von INKEY() zur Unterbrechung von LIST, um zu .berpr.fen, ob der Benutzer die Taste [Esc] gedr.ckt hat:

```
#define K_ESC 27
USE Verkauf NEW
```

```
LIST Filiale, Verkaeufer, Umsatz FOR INKEY() <> K_ESC
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

LOCATE Sucht sequentiell nach einem Datensatz, der die angegebene Bedingung erf.llt

---

Syntax

LOCATE [<Wirkungsbereich>] FOR <lBedingung> [WHILE <lBedingung>]

Argumente

<Wirkungsbereich> spezifiziert den Teil der Datens.tze in der aktuellen Datenbank, der durchsucht werden soll. Die Vorgabe umfa.t alle Datens.tze (ALL).

FOR <lBedingung> gibt den n.chsten, innerhalb von <Wirkungsbereich> zu suchenden Datensatz an.

WHILE <lBedingung> spezifiziert die Gruppe der Datens.tze, die sich vom aktuellen bis zum ersten, die Bedingung nicht erf.llenden Datensatz erstreckt.

Anwendung

LOCATE ist ein Datenbankbefehl, der den ersten Datensatz im aktuellen Arbeitsbereich sucht, der die angegebenen Bedingungen erf.llt und im <Wirkungsbereich> liegt. Wenn Sie LOCATE zum ersten Mal aufrufen, wird beginnend mit dem ersten Datensatz im <Wirkungsbereich> nach dem ersten zutreffenden Datensatz gesucht. LOCATE endet, wenn ein passender Datensatz gefunden oder das Ende von <Wirkungsbereich> erreicht worden ist. Bei erfolgreicher Suche wird der gefundene Datensatz zum aktuellen Datensatz und FOUND() gibt true (.T.) zur.ck. Bei erfolgloser Suche gibt FOUND() false (.F.) zur.ck. Der Datensatzzeiger wird dann abh.ngig von <Wirkungsbereich> positioniert.

Jeder Arbeitsbereich kann .ber eine eigene LOCATE-Bedingung verf.gen. Diese Bedingung bleibt so lange aktiv, bis ein anderer LOCATE-Befehl ausgef.hrt oder die Datei geschlossen worden ist.

LOCATE arbeitet mit CONTINUE zusammen. Nach der Durchf.hrung eines LOCATE kann die Suche mit CONTINUE ab der aktuellen Position des Datensatzzeigers fortgef.hrt werden. Es gibt hierbei aber einige Ausnahmen, wie die folgenden Hinweise zeigen.

Hinweise

. CONTINUE: Die urspr.nglich bei LOCATE angegebenen Argumente f.r den <Wirkungsbereich> und die WHILE-Bedingung werden von CONTINUE ignoriert. Nur eine urspr.nglich angegebene FOR-Bedingung wird ber.cksichtigt. Wenn Sie ein LOCATE unter Einbeziehung einer WHILE-Bedingung fortsetzen m.chten, m.ssen Sie zuerst ein SKIP ausf.hren und dann die originale LOCATE-Anweisung unter Angabe von REST f.r den <Wirkungsbereich> wiederholen. CONTINUE kann hierf.r nicht verwendet werden.

Beispiele

. Diese Beispiele zeigen typische Anwendungen f.r LOCATE:

```
USE Verkauf INDEX Verkauf NEW
LOCATE FOR Filiale = "200"
? FOUND(), EOF(), RECNO()   // Ergebnis: .T. .F. 5
LOCATE FOR Filiale = "5000"
? FOUND(), EOF(), RECNO()   // Ergebnis: .F. .T. 85
```

. In diesem Beispiel wird die Suche mit LOCATE und einer WHILE-Bedingung .ber LOCATE REST fortgef.hrt:

```
SEEK "Winzig"
LOCATE FOR Filiale = "200" WHILE Verkaeufer = "Winzig"
DO WHILE FOUND()
  ? Filiale, Verkaeufer
  SKIP
  LOCATE REST FOR Filiale = "200" WHILE Verkaeufer = "Winzig"
ENDDO
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

MENU TO Aktiviert ein Light-bar-Men. mit vorher definiertem Prompt

---

Syntax

MENU TO <idVar>

Argumente

<idVar> gibt die Variable an, der die Nummer des ausgew.hlten Men.punktes zugewiesen werden soll. Wenn diese Variable nicht existiert oder nicht sichtbar ist, wird eine PRIVATE-Variable angelegt, der die Auswahlnummer zugewiesen wird.

Anwendung

MENU TO ist der Auswahlvorgang f.r das Light-bar-Men.system von Clipper. Vor dem Aufruf von MENU TO m.ssen Sie die Men.punkte und deren zugeh.rige Meldungen .ber die entsprechenden @...PROMPT-Befehle definieren. Der Auswahlvorgang wird mit MENU TO <idVar> aktiviert. Wenn diese Variable nicht existiert oder nicht sichtbar ist, wird eine PRIVATE-Variable angelegt und der erste Men.punkt markiert (optisch hervorgehoben). Ist <idVar> bereits vorhanden, bestimmt der (numerische) Wert dieser Variablen den anf.nglich markierten Men.punkt.

Hinweise

. Farbe: Die Men.punkte werden in der aktuellen Farbe standard angezeigt, w.hrend der jeweils markierte Men.punkt in der Farbe enhanced dargestellt wird.

. Steuerung und Auswahl: Die Cursortasten bewegen die Markierung auf den folgenden oder vorhergehenden Men.punkt. Ist ein Men.punkt markiert, wird die zugeh.rige Meldung in der mit SET MESSAGE festgelegten Zeile angezeigt. Falls Sie SET WRAP auf ON eingestellt haben, springt die Marke vom ersten auf den letzten Men.punkt, wenn an dieser Stelle [.] gedr.ckt worden ist. Entsprechend entgegengesetzt verh.lt sich die Positionierung der Marke, wenn [.] bei Markierung des letzten Men.punktes gedr.ckt wird.
Die Auswahl erfolgt durch Dr.cken von [Return] oder des ersten Buchstabens eines Men.punktes. MENU TO .bergibt dann die Position des ausgew.hlten Men.elementes als numerischen Wert an <idVar>. Wenn Sie [Esc] dr.cken, wird der Auswahlvorgang abgebrochen und Null zur.ckgegeben. Die folgende Tabelle f.hrt alle in MENU TO aktiven Tasten auf.

. SET KEY-Prozeduren: Es ist problemlos m.glich, aus einem aktuellen Auswahlvorgang heraus weitere, ineinander verschachtelte Men.auswahlen zu aktivieren (z.B. .ber eine SET KEY-Prozedur). Alle ausstehenden PROMPTs und MENU TO bleiben erhalten.

Tabelle 4-8: Aktive Tasten in MENU TO

Taste	Aktion
PfeilAufw.rts	Springt zum vorherigen Men.punkt
PfeilAbw.rts	Springt zum folgenden Men.punkt
Pos1	Springt zum ersten Men.punkt
Ende	Springt zum letzten Men.punkt
LinksPfeil	Springt zum vorherigen Men.punkt
RechtsPfeil	Springt zum folgenden Men.punkt
Bild.	W.hlt aktuellen Men.punkt und gibt dessen Position zur.ck
Bild.	W.hlt aktuellen Men.punkt und gibt dessen Position zur.ck
Return	W.hlt aktuellen Men.punkt und gibt dessen Position zur.ck
Esc	Bricht den Auswahlvorgang ab und gibt Null zur.ck
Erster Buchstabe	W.hlt ersten Men.punkt mit dem gleichen Anfangsbuchstaben und gibt dessen Position zur.ck

Beispiele

In diesem Beispiel wird ein einfaches, vertikales Light-bar-Men. mit Meldungen erstellt, die in Zeile 23 zentriert ausgegeben werden. Nach dem Men.aufruf steht der Cursor auf Men.punkt 2, weil nWahl den Anfangswert 2 hat.

```
  LOCAL nWahl := 2
  SET WRAP ON
  SET MESSAGE TO 23 CENTER
  @ 6, 10 PROMPT "Neu " MESSAGE "Neuer Datensatz"
  @ 7, 10 PROMPT "Edit" MESSAGE ".nderung des Datensatzes"
  @ 9, 10 PROMPT "Ende" MESSAGE "R.ckkehr zu DOS"
  MENU TO nWahl
  //
  DO CASE
  CASE nWahl = 0
    QUIT
  CASE nWahl = 1
    NeuerSatz()
  CASE nWahl = 2
    EditSatz()
  CASE nWahl = 3
    QUIT
  ENDCASE
  RETURN
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

NOTE* F.gt eine Kommentarzeile in eine Programmdatei ein

Syntax

NOTE [<KommentarText>]

Argumente

<KommentarText> ist eine Zeichenkette, die nach dem NOTE-Befehl angegeben wird.

Anwendung

NOTE ist ein Synonym f.r das Sternchen-Zeichen, das eine einzelne Kommentarzeile kennzeichnet. Alle nach NOTE folgenden Zeichen werden ignoriert, bis Clipper ein Zeilenende erreicht (Carriage Return/Line Feed). Einzelne Kommentarzeilen k.nnen daher nicht mit einem Semikolon in der n.chsten Zeile fortgesetzt werden.

Einen mehrzeiligen Kommentar k.nnen Sie unter Clipper, .hnlich wie in C, durch die Zeichen /* und */ eingrenzen und kennzeichnen. Wenn Sie einen Teil des Programmcodes, in dem Kommentarzeilen enthalten sind, von der Compilierung ausklammern m.chten, m.ssen Sie die Compiler-Direktiven #ifdef...#endif zur bedingten Compilierung verwenden. Die Verschachtelung von mehreren Kommentaren ist nicht m.glich und nicht erlaubt.

NOTE ist ein Kompatibilit.ts-Befehl und wird daher nicht f.r den Einsatz in neuen Applikationen empfohlen. Er wird durch die aus C .bernommenen Kommentarsymbole /* und */ sowie durch das Zeichen // ersetzt. Au.erdem k.nnen die Kommentarsymbole * und && aus fr.heren Versionen verwendet werden. Eine genaue Erl.uterung der Kommentarsymbole finden Sie in diesem Handbuch im Kapitel Grundkonzepte.

Beispiele

Das folgende Beispiel demonstriert den Einsatz von verschiedenen, in Clipper g.ltigen Kommentarsymbolen:

```
// Dies ist ein Kommentar
/* Dies ist ein Kommentar */
* Dies ist ein Kommentar
&& Dies ist ein Kommentar
NOTE Dies ist ein Kommentar
```

Dateien

PACK Entfernt als gel.scht markierte Datens.tze physisch aus der selektierten Datenbank

---

Syntax

PACK

Anwendung

PACK ist ein Datenbankbefehl, der alle zur L.schung markierten Datens.tze aus der aktuellen Datenbank physisch entfernt, die zugeh.rigen Sektoren auf der Platte als wieder verf.gbar kennzeichnet und alle im aktuellen Arbeitsbereich aktiven Indizes re-indiziert. PACK erzeugt keine tempor.ren Backup-Dateien, die mitvorgenommene Re-Indizierung kann tempor.re Dateien anlegen. Nach Ausf.hrung von PACK wird der Datensatzzeiger auf den ersten logischen Satz des aktuellen Arbeitsbereiches gesetzt.

Bei der Durchf.hrung von PACK mu. die komplette Datenbank in einer Netzwerkumgebung zuerst mit FLOCK() oder USE EXCLUSIVE gesperrt worden sein. Anderenfalls tritt ein Laufzeitfehler auf.

Die Ausf.hrung von PACK kann bei gro.en Datenbanken sehr zeitaufwendig sein. Sie k.nnen aber durch .nderung der Applikation die physische L.schung von Datens.tze eliminieren. Dies erreichen Sie z.B., indem Sie als gel.scht markierte Datens.tze bei Ausgaben, Bearbeitungsfunktionen etc. herausfiltern und bei Neueingaben mit den neuen Daten .berschreiben. Weitere Informationen zur Programmierung von Netzwerkanwendungen finden Sie im Handbuch Programmierung und Hilfsprogramme.

Beispiele

Dieses Beispiel zeigt das Ergebnis eines einfachen PACK-Vorganges:

```
USE Verkauf NEW
? LASTREC()     // Ergebnis: 84
//
DELETE RECORD 4
PACK
? LASTREC()     // Ergebnis: 83
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

QUIT Beendet die Ausf.hrung des laufenden Clipper-Programmes

Syntax

QUIT | CANCEL*

Anwendung

QUIT und CANCEL beenden das laufende Programm, schlie.en alle offenen Dateien und .bergeben die Kontrolle zur.ck an das Betriebssystem. Beide Befehle k.nnen an jeder beliebigen Stelle eines Clipper-Programmes eingesetzt werden. Ein RETURN-Befehl, ausgef.hrt in der obersten Prozedurebene, hat die gleiche Wirkung.

Hinweise

. R.ckgabewert: Wird ein Clipper-Programm durch einen schwerwiegenden Fehler beendet, wird der DOS-ERRORLEVEL auf 1 gesetzt. Wird das Programm hingegen normal beendet, ist dieser R.ckgabewert 0 oder aber die letzte .ber ERRORLEVEL() vorgenommene Einstellung.

Beispiele

Das folgende Beispiel demonstriert den Einsatz von QUIT in einer Dialog-Box:

```
IF DialogJaNein(10, 10, "Zur.ck zu DOS", "BG+/B,B/W", 2)
  QUIT
ENDIF
RETURN
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

READ Aktiviert den Full-screen-.nderungsmodus unter Verwendung von GET-Objekten

---

Syntax

READ [SAVE]

Argumente

SAVE sichert den Inhalt des aktuellen GetList-Arrays, nachdem READ beendet worden ist. Im weiteren Ablauf des Programmes k.nnen Sie dieselben GET-Objekte durch Angabe eines weiteren READ-Befehles bearbeiten. Wenn SAVE nicht angegeben wird, bekommt die aktuelle GetList-Variable ein leeres Array zugewiesen, wobei die vorherigen GET-Objekte mit dem Ende von READ gel.scht werden.

Anwendung

READ aktiviert einen Full-screen-Bearbeitungsmodus, der all diejenigen GET-Objekte verwendet, die seit Ausf.hrung von CLEAR, CLEAR GETS, CLEAR ALL oder READ erzeugt oder dem aktuellen GetList-Array hinzugef.gt worden sind. Falls eine aktive Format-Prozedur existiert, wird diese Prozedur vor dem Full-screen-Bearbeitungsmodus ausgef.hrt.

W.hrend eines READ k.nnen Sie den Puffer jedes einzelnen GET-Objektes .ndern und sich von Objekt zu Objekt bewegen. Falls Sie f.r ein GET-Objekt .ber die WHEN-Klausel eine vorher auszuwertende <lPre-Bedingung> spezifiziert haben, gelangen Sie nur bei Erf.llung dieser Bedingung in das zugeh.rige GET-Eingabefeld. Ist das Ergebnis von <lPre-Bedingung> false (.F.), verzweigt die Kontrolle zum n.chsten GET-Objekt in GetList. Die Bearbeitung eines GET-Puffers kann unter Benutzung aller Editier- und Steuertasten erfolgen (vgl. folgende Tabellen).

Wenn der Benutzer eine GET-Abschlu.taste gedr.ckt hat, wird anschlie.end eine eventuell angegebene RANGE- oder VALID-Klausel ausgewertet. Ist das Ergebnis dieser Klausel true (.T.), verzweigt die Kontrolle zum n.chsten GET-Objekt. Ist das Ergebnis aber false (.F.), verbleibt die Kontrolle so lange beim aktuellen GET-Objekt, bis ein g.ltiger (d.h. die Klausel erf.llender) Wert eingegeben oder [Esc] bet.tigt worden ist.

Wenn der Benutzer in ein GET-Objekt einen g.ltigen Wert eingegeben hat, wird der Inhalt des GET-Puffers der zugeh.rigen Variablen zugewiesen.

Die folgenden Tabellen geben die w.hrend eines READ-Vorganges aktiven Tasten an:

Tabelle 4-9: Steuertasten in READ

Taste	Aktion
LinksPfeil, Strg+S	N.chstes Zeichen links, kein Sprung in das vorherige GET
RechtsPfeil, Strg+D	N.chstes Zeichen rechts, kein Sprung in das n.chste GET
Strg+LinksPfeil, Strg+A	N.chstes Wort links
Strg+RechtsPfeil, Strg+F	N.chstes Wort rechts
PfeilAufw.rts, Strg+E	Vorheriges GET
PfeilAbw.rts, Strg+X, Return,Strg+M	N.chstes GET
Pos1	Erstes Zeichen vom GET
Ende	Letztes Zeichen vom GET
Strg+Pos1	Beginn des ersten GET
Strg+Ende	Beginn des letzten GET

Tabelle 4-10: Editiertasten in READ

Taste	Aktion
Entf, Strg+G	L.scht das aktuelle Zeichen unter dem Cursor
R.cktaste, Strg+H	L.scht das Zeichen links vom Cursor
Strg+T	L.scht das n.chste Wort rechts
Strg+Y	L.scht ab der Cursor-Position bis zum GET-Ende
Strg+U	Speichert den Originalwert in das aktuelle GET zur.ck

Tabelle 4-11: Umschalttasten in READ

Taste	Aktion
Einfg, Strg+G	Schaltet den Einf.gemodus ein/aus

Tabelle 4-12: Abschlu.tasten in READ

Taste	Aktion
Strg+W, Strg+C, Bild., Bild.	Beendet READ und sichert aktuelles GET
Return, Strg+M	Beendet READ vom letzten GET
Esc	Beendet READ und sichert aktuelles GET nicht
PfeilAufw.rts	Beendet READ vom ersten GET, wenn READEXIT() = .T.
PfeilAbw.rts	Beendet READ vom letzten GET, wenn READEXIT() = .T.

Hinweise

. Verschachtelte READs: Wenn Sie ein verschachteltes READ aus einer SET KEY-, VALID- oder WHEN-Funktion bzw. aus einer UDF heraus durchf.hren m.chten, m.ssen Sie zuerst ein neues GetList-Array deklarieren oder erzeugen. Danach f.hren Sie die entsprechenden @...SAY-Befehle und ein READ aus. Nach Abschlu. dieses READs wird das neue GetList-Array gel.scht und das alte GetList-Array wird wieder sichtbar. Ein Beispiel hierf.r finden Sie weiter unten.

. Quick keys: Durch wiederholtes Dr.cken von [Pos1] oder [Ende] springt der Cursor auf den ersten bzw. letzten Buchstaben im GET-Objekt-Puffer, der kein Leerzeichen ist.

. Beendigung von READ: Ein READ wird durch Ausf.hrung von BREAK, CLEAR, CLEAR GETS oder CLEAR ALL aus einer SET KEY-Prozedur oder VALID-UDF heraus beendet.

. UPDATED(): Wenn der Inhalt eines beliebigen GET-Objekt-Puffers w.hrend des aktuellen READ-Vorganges ge.ndert worden ist, wird UPDATED() auf true (.T.) gesetzt.

Beispiele

. In diesem Beispiel werden mehrere GETs definiert und mit READ bearbeitet:

```
CLEAR
cVar1 := cVar2 := cVar3 := SPACE(10)
@ 10, 10 SAY "Variable eins:" GET cVar1 VALID .NOT. EMPTY(cVar1)
@ 11, 10 SAY "Variable zwei:" GET cVar2 WHEN RTRIM(cVar1) <> Eins"
@ 12, 10 SAY "Variable drei:" GET cVar3 VALID .NOT. EMPTY(cVar3)
READ
```

. Das folgende Beispiel demonstriert ein verschachteltes READ:

```
LOCAL cName := SPACE(10)
@ 10, 10 GET cName VALID SubForm(cName)
READ
RETURN
FUNCTION SubForm(cLookup)
  LOCAL GetList := {}            // Erzeuge neues GetList
  USE Verkauf INDEX Verkauf NEW
  SEEK cLookup
  IF FOUND()
    @ 15, 10 GET Verkaeufer      // Neue GET-Objekte f.r GetList
    @ 16, 10 GET Umsatz
    READ                         // READ von neuem GetList
  ENDIF
  CLOSE Verkauf
RETURN .T.                       // Freigabe von GetList
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

RECALL Hebt die Markierung von als gel.scht gekennzeichneten Datens.tzen auf

---

Syntax

RECALL [<Wirkungsbereich>] [WHILE <lBedingung>] [FOR <lBedingung>]

Argumente

<Wirkungsbereich> spezifiziert den Teil, innerhalb dessen Markierungen in der aktuellen Datenbank zur.ckgesetzt werden sollen. Vorgegeben ist nur der jeweils aktuelle Datensatz NEXT 1. Wurde jedoch eine Bedingung als Klausel angegeben, umfa.t die Vorgabe alle Datens.tze (ALL).

WHILE <lBedingung> spezifiziert die Gruppe der Datens.tze, die sich vom aktuellen bis zum ersten, die Bedingung nicht erf.llenden Datensatz erstreckt.

FOR <lBedingung> selektiert alle Datens.tze innerhalb von <Wirkungsbereich>, welche die angegebene Bedingung erf.llen und deren Markierung somit zur.cksetzt.

Anwendung

RECALL ist ein Datenbankbefehl, der die L.schmarken der Datens.tze im aktuellen Arbeitsbereich aufhebt. RECALL stellt die Umkehrung von DELETE dar. Wenn DELETED auf ON eingestellt ist, kann RECALL die L.schmarkierung des aktuellen oder eines bestimmten, mit RECORD <nDatensatz> angegebenen Datensatzes zur.cksetzen. Wurde eine Datei aber bereits mit PACK bearbeitet, sind alle vorher als gel.scht markierten Datens.tze physisch entfernt worden und k.nnen auf keinen Fall mehr zur.ckgeholt werden.

In einer Netzwerkumgebung mu. der aktuelle Datensatz zuerst mit RLOCK() gesperrt werden, bevor RECALL durchgef.hrt werden kann. Bei der gleichzeitigen Bearbeitung von mehreren Datens.tzen mu. zuerst die komplette Datenbank mit FLOCK() oder USE EXCLUSIVE gesperrt worden sein. Weitere Informationen zur Programmierung von Netzwerkanwendungen finden Sie im Handbuch Programmierung und Hilfsprogramme.

Beispiele

Dieses Beispiel zeigt einige Ergebnisse von RECALL:

```
USE Verkauf NEW
//
DELETE RECORD 4
? DELETED()      // Ergebnis: .T.
//
RECALL
? DELETED()      // Ergebnis: .F.
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

REINDEX Baut alle ge.ffneten Indexdateien des selektierten Arbeits-
bereiches neu auf

Syntax

REINDEX

Anwendung

REINDEX ist ein Datenbankbefehl, der alle ge.ffneten Indizes im aktuellen Arbeitsbereich komplett neu aufbaut. Nach Beendigung des Reindizierungs-Prozesses bleiben alle erneuerten Indexdateien ge.ffnet, SET ORDER wird auf 1 gesetzt, und der Datensatzzeiger steht auf dem ersten Datensatz des kontrollierenden Indexes. Wurde ein Index mit SET UNIQUE ON oder INDEX...UNIQUE erzeugt, wird auch bei der Reindizierung nur der jeweils erste von mehreren gleichen Schl.sseln angeh.ngt.

In einer Netzwerkumgebung erfordert REINDEX, da. die aktuelle Datenbank mit USE EXCLUSIVE ge.ffnet worden ist. Weitere Informationen zur Programmierung von Netzwerkanwendungen finden Sie im Handbuch Programmierung und Hilfsprogramme.

Achtung

REINDEX erzeugt f.r die jeweils neu aufzubauende Indexdatei keinen neuen Header. Daher kann REINDEX keine Fehler im Datei-Header beseitigen. Ein g.ltiger, fehlerfreier Index kann nur mit dem Befehl INDEX ON neu erzeugt werden.

Beispiele

Dieses Beispiel zeigt die Reindizierung von Verkauf.DBF:

```
USE Verkauf INDEX Verkauf NEW, Gebiet NEW
REINDEX
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

RELEASE L.scht PUBLIC- und PRIVATE-Speichervariablen

Syntax

RELEASE <idSpeichervariablenListe>
RELEASE ALL [LIKE | EXCEPT <VariablenSpezifikation>]

Argumente

<idSpeichervariablenListe> ist eine Liste der zu l.schenden PRIVATE- oder PUBLIC-Variablen und/oder -Arrays.

ALL [LIKE | EXCEPT <VariablenSpezifikation>] definiert eine Gruppe von sichtbaren PRIVATE-Speichervariablen, denen NIL zugewiesen (LIKE) bzw. denen NIL nicht zugewiesen werden soll (EXCEPT). Dabei gibt <VariablenSpezifikation> die entsprechenden Speichervariablen an. Wildcards (* und ?) sind hierbei zugelassen.

Anwendung

RELEASE ist ein Speichervariablen-Befehl, der je nach Angabe eine von zwei Aufgaben erf.llt. Wird RELEASE zusammen mit einer <idSpeichervariablenListe> angegeben, werden die spezifizierten PRIVATE- oder PUBLIC-Variablen und/oder -Arrays aus dem Arbeitsspeicher entfernt.

Wurde RELEASE mit einer der ALL-Klauseln aufgerufen, wird den in der laufenden Prozedur erzeugten PRIVATE-Speichervariablen NIL zugewiesen. Diese Variablen werden erst endg.ltig aus dem Arbeitsspeicher entfernt, wenn die Erzeuger-Prozedur oder -UDF beendet wird. Mit der ALL-Klausel k.nnen keine PUBLIC-Variablen gel.scht werden. Sie m.ssen entweder explizit Name f.r Name oder mit CLEAR MEMORY gel.scht werden.

RELEASE wirkt nicht auf LOCAL- oder STATIC-Variablen. LOCAL-Variablen werden automatisch mit Beendigung der Prozedur oder UDF gel.scht, in denen sie deklariert worden sind. STATIC-Variablen k.nnen nicht gel.scht werden, da sie w.hrend des gesamten Programmablaufs existieren.

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

RENAME Benennt eine existierende Datei um

Syntax

RENAME <xcAlteDatei> TO <xcNeueDatei>

Argumente

<xcAlteDatei> ist der Name der umzubennenden Datei. Der Name kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Die Erweiterung des Dateinamens mu. mit angegeben werden. Falls erforderlich, k.nnen Sie auch das Laufwerk und/oder den Pfad angeben.

<xcNeueDatei> ist der Name der neuen Datei. Der Name kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Die Erweiterung des Dateinamens mu. mit angegeben werden. Falls erforderlich, k.nnen Sie auch das Laufwerk und/oder den Pfad angeben.

Anwendung

RENAME ist ein Dateibefehl, der den Namen der angegebenen Datei in einen neuen Namen umwandelt. Wenn Quell- und Zielpfad verschieden sind, wird die komplette Datei in den Zielpfad verlegt. RENAME verwendet nicht die Einstellungen von SET DEFAULT und SET PATH, um nach <xcAlteDatei> zu suchen. Daher wird <xcAlteDatei> nur umbenannt, wenn sie sich im aktuellen DOS-Verzeichnis oder im angegebenen Pfad befindet.

Falls <xcNeueDatei> bereits existiert oder augenblicklich ge.ffnet ist, nimmt RENAME keine Umbenennung vor. Zur Abfrage dieser Bedingungen (z.B. zur Fehlerbehandlung) k.nnen Sie vor Aufruf von RENAME die Funktion FILE() verwenden (vgl. nachfolgendes Beispiel).

Achtung

Vor einer Umbenennung mu. die jeweilige Datei geschlossen sein. Der Versuch, eine ge.ffnete Datei umzubenennen, kann unvorhersehbare Folgen haben. Weiterhin sollte beachtet werden, da. bei Umbenennung einer Datenbank auch die zugeh.rige .DBT-Datei den neuen Namen erhalten mu.. Wenn Sie dies unterlassen, gef.hrden Sie die Sicherheit des betreffenden Datenbestandes.

Beispiele

Dieses Beispiel zeigt, wie vor Ausf.hrung des RENAME-Befehles das eventuelle Vorhandensein der Zieldatei .berpr.ft wird:

```
xcAlteDatei := "AltDatei.TXT"
xcNeueDatei := "NeuDatei.TXT"
IF .NOT. FILE(xcNeueDatei)
  RENAME (xcAlteDatei) TO (xcNeueDatei)
ELSE
  ? "Zieldatei ist bereits vorhanden"
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

REPLACE Weist Datenbankfeldern neue Werte zu

Syntax

REPLACE <idFeld> WITH <Ausdruck> [,<idFeld2> WITH <Ausdruck2>...]
[<Wirkungsbereich>] [WHILE <lBedingung>] [FOR <lBedingung>]

Argumente

<idFeld> gibt den Namen einer vorhandenen Feldvariablen an, der ein neuer Wert zugewiesen werden soll. Wurde <idFeld> mit einem Alias-Verweis gekennzeichnet, wird auf den entsprechenden Alias-Arbeitsbereich zugegriffen.

<Ausdruck> ist der neue Wert, der <idFeld> zugewiesen werden soll.

<Wirkungsbereich> spezifiziert den Teil, der in der aktuellen Datenbank ersetzt werden soll. Vorgegeben ist der jeweils aktuelle Datensatz NEXT 1. Wurde eine Bedingung angegeben, umfa.t die Vorgabe alle Datens.tze (ALL) im aktuellen Arbeitsbereich.

WHILE <lBedingung> spezifiziert die Gruppe der Datens.tze, die sich vom aktuellen bis zum ersten, die Bedingung nicht erf.llenden Datensatz erstreckt.

FOR <lBedingung> selektiert alle Datens.tze innerhalb des <Wirkungsbereiches>, welche die angef.hrte Bedingung erf.llen und somit ersetzt werden.

Anwendung

REPLACE ist ein Datenbankbefehl, der einem oder mehreren Datenbankfeldern in den angegebenen Arbeitsbereichen neue Werte zuweist. Die Zielfelder k.nnen vom Typ Zeichen, Datum, numerisch, logisch oder Memo sein. REPLACE f.hrt dieselbe Funktion wie der Zuweisungsoperator := aus, mit dem Unterschied, da. nicht mit einem Alias-Verweis gekennzeichnete Variablennamen auf Datenbankfelder bezogen werden. Das bedeutet, da. Sie Datenbankfeldern auch neue Werte .ber andere Zuweisungsbefehle zuordnen k.nnen, vorausgesetzt, dem Feldnamen ist ein Alias-Verweis oder FIELD-Alias vorangestellt oder die Feldvariable ist als FIELD deklariert worden.

Der vorgegebene <Wirkungsbereich> f.r REPLACE ist der aktuelle Datensatz, au.er es ist eine Bedingung oder ein anderer <Wirkungsbereich> angegeben. In letzterem Fall wird die REPLACE-Aktion f.r jeden Datensatz durchgef.hrt, der die Bedingung erf.llt oder der in den <Wirkungsbereich> f.llt.

Achtung

Wenn Sie mit REPLACE ein Schl.sselfeld ersetzen, wird der Index aktualisiert und die relative Position des Datensatzzeigers innerhalb des Indexes ver.ndert. Es k.nnen sich daher Fehler ergeben, wenn ein Schl.sselfeld aktualisiert wird und eine Bedingung oder ein <Wirkungsbereich> angegeben war. Die L.sung f.r dieses Problem besteht darin, da. Sie vor dem REPLACE den Befehl SET ORDER TO 0 ausf.hren. Nun erfolgt der REPLACE-Vorgang, auch von Schl.sselfeldern, in der nat.rlichen Reihenfolge der Datenbank. Da die Indexdateien weiterhin offenbleiben, werden auch sie aktualisiert, wenn der Wert eines Schl.sselfeldes ersetzt wird.

In einer Netzwerkumgebung mu. der aktuelle Datensatz zuerst mit RLOCK() gesperrt werden, bevor REPLACE durchgef.hrt werden kann. Bei der gleichzeitigen Bearbeitung von mehreren Datens.tzen .ber eine Bedingung oder einen <Wirkungsbereich> mu. zuerst die komplette Datenbank mit FLOCK() oder USE EXCLUSIVE gesperrt worden sein. Der Datensatz eines Datenbankfeldes, das .ber einen Alias-Verweis in einem anderen Arbeitsbereich ersetzt wird, mu. ebenfalls mit RLOCK() gesperrt worden sein. Weitere Informationen zur Programmierung von Netzwerkanwendungen finden Sie im Handbuch Programmierung und Hilfsprogramme.

Beispiele

. Dieses Beispiel zeigt eine einfache Anwendung von REPLACE:

```
USE Kunden NEW
APPEND BLANK
USE Rechnung NEW
APPEND BLANK
//
REPLACE RgDatum WITH DATE(), KundenNr WITH Kunden->KundenNr,;
  Kunden->LetzteRg WITH DATE()
```

. Dieselben Zuweisungen k.nnen alternativ mit folgenden Anweisungen vorgenommen werden:

```
FIELD->RgDatum  := DATE()
FIELD->KundenNr := Kunden->KundenNr
Kunden-LetzteRg := DATE()
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

REPORT FORM Gibt einen Report auf der Ausgabeeinheit aus

Syntax

```
REPORT FORM <xcReportDatei>
  [TO PRINTER] [TO FILE <xcDatei>]
  [<Wirkungsbereich>] [WHILE <lBedingung>] [FOR <lBedingung>]
  [PLAIN | HEADING <cKopfzeile>] [NOEJECT] [SUMMARY]
```

Argumente

<xcReportDatei> ist der Name der .FRM-Datei, in der das Layout des Reportes abgelegt ist. Der Name kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Falls keine andere Erweiterung angegeben ist, wird die Standard-Erweiterung .FRM benutzt.

TO PRINTER f.hrt die Ausgabe zus.tzlich auf dem Drucker aus.

TO FILE leitet die Ausgabe zus.tzlich in die Datei <xcDatei> um, wobei keine Steuerzeichen f.r den Seitenvorschub (ASCII 12) .bernommen werden. <xcDatei> kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Falls keine andere Erweiterung angegeben ist, wird die Standard-Erweiterung .TXT hinzugef.gt.

<Wirkungsbereich> spezifiziert den Teil der Datens.tze in der aktuellen Datenbank, der als Bericht ausgegeben werden soll. Die Vorgabe umfa.t alle Datens.tze (ALL).

WHILE <lBedingung> spezifiziert die Gruppe der Datens.tze, die sich vom aktuellen bis zum ersten, die Bedingung nicht erf.llenden Datensatz erstreckt.

FOR <lBedingung> selektiert alle Datens.tze innerhalb des <Wirkungsbereiches>, welche die angef.hrte Bedingung erf.llen und somit im Report ber.cksichtigt werden.

PLAIN unterdr.ckt die Ausgabe des Datums und der Seitennummer. Au.erdem wird der Report durchgehend ohne Seitenumbruch ausgegeben, wobei die Report- und Spalten.berschriften nur am Reportanfang stehen.

HEADING gibt das Ergebnis von <cKopfzeile> in der ersten Zeile jeder Seite aus. <cKopfzeile> wird nur einmalig ganz zu Beginn der Reportausgabe, bevor der Datensatzzeiger bewegt wird, ausgewertet. Falls PLAIN und HEADING angegeben werden, hat PLAIN Vorrang.

NOEJECT unterdr.ckt bei Verwendung der Klausel TO PRINTER den Seitenvorschub vor Druckbeginn.

SUMMARY bewirkt, da. REPORT FORM nur die Gruppen-, Untergruppen- und Gesamtsummen-Zeilen ausgibt. Die dazwischen liegenden Zeilen mit den dazugeh.rigen Einzelpositionen werden unterdr.ckt.

Anwendung

REPORT FORM ist ein Konsolen-Befehl, der die Datens.tze im aktuellen Arbeitsbereich sequentiell verarbeitet und sie in einem tabellarischen, optional untergliederten Bericht mit Seiten- und Spalten.berschriften entsprechend des in <xcReportDatei> abgelegten Layouts ausgibt. M.glichkeiten zur Erzeugung einer <xcReportDatei> sind z.B. durch das Hilfsprogramm RL.EXE gegeben. Weitere Informationen zur Erstellung von Report-Layouts finden Sie im Handbuch Programmierung und Hilfsprogramme im Kapitel Reports und Labels.

Nach Aufruf von REPORT FORM wird die Ausgabe an den Bildschirm und optional an den Drucker oder in eine Datei gesendet. Damit w.hrend der Ausgabe auf einen Drucker oder in eine Datei keine Bildschirmausgabe erfolgt, m.ssen Sie vorher SET CONSOLE auf OFF setzen.

REPORT FORM sucht die <xcReportDatei> zuerst im aktuellen Verzeichnis, soweit kein anderer Pfad mit angegeben worden ist. Wurde die Datei nicht

gefunden, wird im durch SET PATH angegebenen Pfad gesucht.

Hinweise

. Unterbrechung der Ausgabe: Um dem Benutzer eine Unterbrechung der REPORT FORM-Ausgabe zu erlauben, geben Sie bei der FOR-Bedingung eine Tastenabfrage und -.berpr.fung mit INKEY() an (vgl. Beispiel).

. Druckrand: REPORT FORM ber.cksichtigt, wie jeder Konsolen-Befehl, die aktuelle Einstellung von SET MARGIN, wenn die Ausgabe auf dem Drucker erfolgt.

. Ausgabe von Seitenvorsch.ben in eine Zieldatei: Die bei Angabe von der Klausel TO FILE beschriebene Unterdr.ckung von Seitenvorsch.ben kann umgangen werden, indem Sie die Druckausgabe .ber SET PRINTER TO in eine Datei umleiten. Dazu ein Beispiel:

```
SET PRINTER TO <xcDatei>
REPORT FORM <xcReportDatei> TO PRINTER
SET PRINTER TO
```

. REPORT in einer Netzwerkumgebung: Bei Ausf.hrung des REPORT FORM-Befehles k.nnen .nderungen an Datens.tzen, die andere Netzwerkanwender w.hrend der Reportausgabe vornehmen, zu ungewollten Ausgabeergebnissen f.hren. Hat z.B. ein Anwender einen Schl.sselwert von "A" in "Z" ge.ndert, kann dieser Datensatz u.U. zweimal im Report vorkommen, da sich die Indexreihenfolge ge.ndert hat.

Beispiele

. Dieses Beispiel benutzt einmal Literale und einmal einen erweiterten Ausdruck zur Ausgabe des Reports:

```
LOCAL xcReportDatei := "Verkauf"
USE Verkauf INDEX Verkauf NEW
REPORT FORM Verkauf TO PRINTER FOR Filiale = "100";
  HEADING "Filiale 100"
REPORT FORM (xcReportdatei) TO PRINTER FOR Filiale <> "100"
```

. Das folgende Beispiel demonstriert den Einsatz von INKEY() zur Unterbrechung von REPORT FORM, um zu .berpr.fen, ob der Benutzer [Esc] gedr.ckt hat:

```
#define K_ESC 27
USE Verkauf INDEX Verkauf NEW
REPORT FORM Verkauf FOR INKEY() <> K_ESC
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

RESTORE Liest Speichervariablen aus einer .MEM-Datei ein

Syntax

RESTORE FROM <xcMemDatei> [ADDITIVE]

Argumente

<xcMemDatei> gibt die von der Platte zu ladende .MEM-Datei an. Der Name kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Falls keine andere Erweiterung angegeben ist, wird die Standard-Erweiterung .MEM angenommen.

ADDITIVE bewirkt, da. die geladenen Speichervariablen zu den bereits im Arbeitsspeicher vorhandenen Variablen hinzugef.gt werden.

Anwendung

RESTORE ist ein Speichervariablen-Befehl, der vorher mit SAVE in eine .MEM-Datei gesicherte PUBLIC- und PRIVATE-Speichervariablen wiedererzeugt und sie mit ihren fr.heren Werten initialisiert. Der G.ltigkeitsbereich einer Variablen wird mit SAVE nicht abgespeichert. Er wird stattdessen bei Neuanlage der Variablen durch RESTORE neu festgelegt. Arrays und LOCAL-Variablen k.nnen nicht durch SAVE und RESTORE gesichert bzw. zur.ckgeholt werden.

Eine Variable wird bei der Neuanlage durch RESTORE als PRIVATE deklariert, d.h. sie ist nur in der laufenden Prozedur oder UDF und deren Unterprozeduren sichtbar. Wurde allerdings der ADDITIVE-Zusatz angegeben und ist bereits eine PUBLIC-Variable desselben Namens vorhanden, wird der neue Wert dieser existierenden Variablen zugewiesen.
Wird der ADDITIVE-Zusatz nicht angegeben, dann werden zuerst alle vorhandenen PUBLIC- und PRIVATE-Variablen gel.scht. Danach werden die Speichervariablen der .MEM-Datei neu angelegt. Die Angabe von ADDITIVE bewirkt, da. PUBLIC- und PRIVATE-Variablen mit dem gleichen Namen .berschrieben werden, au.er sie wurden mit einer PRIVATE-Deklaration versteckt.

LOCAL- und STATIC-Variablen werden von RESTORE nicht ber.cksichtigt. Ist eine in der laufenden Prozedur deklarierte LOCAL- oder STATIC-Variable vorhanden, die den gleichen Namen wie eine mit RESTORE zur.ckgeholte Speichervariable hat, dann ist grunds.tzlich nur die LOCAL- oder STATIC-Variable sichtbar. Die zur.ckgeholte Speichervariable kann nur .ber den Alias-Verweis MEMVAR angesprochen werden.

Beispiele

Dieses Beispiel zeigt eine typische Anwendung f.r SAVE und RESTORE. Speichervariablen, die den Inhalt von Bildschirmausgaben enthalten, werden in einer .MEM-Datei abgespeichert und wieder geladen:

```
// Im vorherigen Programmablauf ist ein Pseudo-Array mit
// mehreren Bildschirminhalten erzeugt worden
SAVE SCREEN TO cBildschirm1
SAVE ALL LIKE cBildschirm* TO Masken
//
<Anweisungen>...
//
RESTORE FROM Masken ADDITIVE
nNummer := "1"
RESTORE SCREEN FROM ("cBildschirm" + nNummer)
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

RESTORE SCREEN* Gibt einen vorher gespeicherten Bildschirminhalt aus

---

Syntax

RESTORE SCREEN [FROM <cBildschirm>]

Argumente

FROM <cBildschirm> gibt einen Zeichenausdruck an, der in den Bildschirmspeicher geladen werden soll.

Anwendung

RESTORE SCREEN ist der synonyme Befehl f.r die Funktion RESTSCREEN(), die einen vorher gesicherten Bildschirm wieder anzeigt. RESTORE SCREEN wird in Zusammenhang mit SAVE SCREEN verwendet, um z.B. eine Bildschirmmaske wieder anzuzeigen, ohne die evtl. vorher benutzten Befehlsfolgen von @...SAY, @...GET oder ? wieder komplett abzuarbeiten.

RESTORE SCREEN arbeitet auf zwei Arten, abh.ngig davon, ob die FROM-Klausel angegeben worden ist oder nicht. Wurde sie mit angegeben, wird der Bildschirm aus <cBildschirm> zur.ckgesichert. <cBildschirm> ist ein Zeichenausdruck, der normalerweise einen mit SAVE SCREEN zugewiesenen Bildschirminhalt enth.lt. Ist die FROM-Klausel nicht mit angegeben worden, wird der Bildschirm aus dem Vorgabe-Bildschirmpuffer zur.ckgesichert, der bei Aufruf des SAVE SCREEN-Befehles ohne die TO-Klausel erzeugt wird.

Teile eines Bildschirmes k.nnen Sie mit SAVESCREEN() und RESTSCREEN() sichern und zur.ckholen.

RESTORE SCREEN ist ein Kompatibilit.ts-Befehl und wird daher nicht zum Einsatz in neuen Applikationen empfohlen. Er wird durch die Funktion RESTSCREEN() ersetzt, die neben kompletten Bildschirminhalten auch Teile davon wiederherstellen kann.

Achtung
SAVE SCREEN, RESTORE SCREEN, SAVESCREEN() und RESTSCREEN() werden nur bei Verwendung des vorgegebenen Bildschirmtreibers (IBM-PC, memory mapped) unterst.tzt. Andere Bildschirmtreiber unterst.tzen diese M.glichkeiten u.U. nicht.

Beispiele

Das folgende Beispiel gibt eine kleine Pop-Up-Meldungsbox aus, wobei SAVE SCREEN und RESTORE SCREEN Verwendung finden:

```
IF DateiMeldg()
  COPY FILE Quell.TXT TO Ziel.TXT
ELSE
  BREAK
ENDIF
RETURN
//
FUNCTION DateiMeldg
  LOCAL lAntwort := .F., cBildschirm
  SAVE SCREEN TO cBildschirm
  @ 10, 10 CLEAR TO 12, 45
  @ 10, 10 TO 12, 45 DOUBLE
  @ 11, 12 SAY "Datei existiert. .berschreiben? (J/N)" ;
    GET lAntwort PICT "Y"
  READ
  RESTORE SCREEN FROM cBildschirm
RETURN lAntwort
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

RUN F.hrt einen DOS-Befehl oder ein DOS-Programm aus

Syntax

RUN | !* <xcBefehlsZeile>

Argumente

<xcBefehlsZeile> kann jedes ausf.hrbare Programm sein, einschlie.lich COMMAND.COM und residenter DOS-Befehle. Der Name kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden.

Anwendung

RUN f.hrt einen DOS-Befehl oder ein DOS-Programm aus der laufenden Applikation heraus aus, wobei Clipper eine neue Kopie von COMMAND.COM aufruft und dieser gleichzeitig die DOS-Befehlszeile .bergibt. Dadurch ergeben sich zwei Bedingungen:
Erstens mu. gen.gend Speicher f.r COMMAND.COM (etwa 27 KByte f.r DOS 3.3, 37 KByte f.r DOS 4.0) und das auszuf.hrende Programm vorhanden sein, zweitens mu. COMMAND.COM im COMSPEC-Pfad vorhanden sein. Die Vorgabe f.r den COMSPEC-Pfad ist das Hauptverzeichnis des Boot-Laufwerkes. Falls COMMAND.COM nicht in diesem Pfad vorhanden oder die Platte ver.ndert worden ist, m.ssen Sie SET COMSPEC vor Aufruf der Clipper-Applikation entsprechend neu setzen. SET DEFAULT und SET PATH beeinflussen RUN nicht.

Die alternative Form von RUN ist ! und nur noch aus Kompatibilit.tsgr.nden vorhanden, d.h. nicht zum Einsatz in neuen Applikationen empfohlen.

Achtung

Der Aufruf eines speicherresidenten Programmes aus einer Clipper-Applikation heraus kann der Applikation vorher verf.gbaren Speicher entziehen und sollte daher unter keinen Umst.nden vorgenommen werden.

Beispiele

. Das folgende Beispiel demonstriert den Einsatz von RUN in Kombination mit MEMOREAD() und MEMOWRIT(), um eine UDF zu erstellen, die einen externen Texteditor zur Bearbeitung des aktuellen Memo-Feldes aufruft:

```
lErfolg := EditorMemo("QEdit", "Bemerkung")
RETURN
FUNCTION EditorMemo(cEditor, cMemoFeld)
LOCAL lWritten := MEMOWRIT("ClipEdit.Tmp", &cMemoFeld.)
  IF lWritten
    RUN (cEditor + " ClipEdit.Tmp")
    REPLACE &cMemoFeld. WITH MEMOREAD("ClipEdit.Tmp")
    ERASE ClipEdit.Tmp
  ENDIF
RETURN lWritten
```

. Wenn Sie dem Benutzer aus Ihrer Anwendung heraus den direkten Zugriff auf DOS erlauben m.chten, k.nnen Sie den folgenden Befehl einsetzen:

```
RUN COMMAND
```

Damit dem Benutzer die R.ckkehr zu Ihrer Clipper-Applikation erleichtert wird, sollten Sie den DOS-Prompt in einer Batch-Datei zum Aufruf Ihrer Applikation z.B. wie folgt .ndern:

```
REM Batch-Datei f.r Clipper-Applikationen
ECHO OFF
PROMPT Geben Sie EXIT zur R.ckkehr in das Hauptprogramm ein $_$p$g
<Applikationsname>
PROMPT $p$g
```

Der Benutzer mu. dann nat.rlich Ihre Batch-Datei anstelle der .EXE-Datei zum Aufruf Ihrer Applikation verwenden.

Dateien

AEVAL() F.hrt einen Codeblock f.r jedes Element eines Arrays aus

---

Syntax

AEVAL(<aArray>, < bBlock>, [<nStart>], [<nAnzahl>]) -> aArray

Parameter

<aArray> ist das Array, das durchlaufen wird.

<bBlock> ist der Codeblock, der f.r jedes Element ausgef.hrt wird.

<nStart> ist die Position des ersten Elements, mit dem <bBlock> ausgef.hrt wird. Der Standardwert ist das erste Element von <aArray>.

<nAnzahl> ist die Anzahl der Elemente, die ab <nStart> durchlaufen werden sollen. Wird der Parameter nicht angegeben, so wird die Bearbeitung bis einschlie.lich zum letzten Element <bBlock> ausgef.hrt.

R.ckgabe

AEVAL() gibt die Referenz auf <aArray> zur.ck.

Anwendung

AEVAL(), eine Arrayfunktion, ist immer dann zu benutzen, wenn ein Codeblock mit einer Reihe von Elementen eines Arrays ausgef.hrt werden soll. AEVAL() f.hrt f.r jedes .ber <nStart> und <nAnzahl> spezifizierte Element einen Codeblock aus, wobei der Wert des Elements dem Codeblock als Argument .bergeben wird. Der R.ckgabewert des Codeblocks wird ignoriert.

AEVAL() setzt keinen bestimmten Inhalt f.r ein Array-Element voraus. Es wird vorausgesetzt, da. der unterst.tzende Codeblock wei. oder ermittelt, welchen Datentyp jedes Element enth.lt.

AEVAL() arbeitet .hnlich der Funktion DBEVAL(), welche einen Codeblock f.r jeden Datensatz einer Datei verwendet. Wie DBEVAL kann auch AEVAL() als einfaches Mittel dazu benutzt werden, iterative Anweisungen f.r komplexe Arraystrukturen auszuf.hren.

Weitergehende Informationen .ber die Theorie und Syntax von Codebl.cken erhalten Sie im Abschnitt Code Bl.cke des Kapitels Grundkonzepte.

Beispiele

. Dieses Beispiel benutzt die Funktion AEVAL(), um den Inhalt eines Arrays anzuzeigen, welches .ber die DIRECTORY()-Funktion zur.ckgegeben wird. Gleichzeitig werden die Dateigr..en in diesem Array aufaddiert und durch eine weitere Anweisung angezeigt:

```
#include "DIRECTRY.CH"
LOCAL aDateien := DIRECTORY("*.dbf"), nTotal := 0
AEVAL(aDateien,;
  { | aDatei | QOUT(PADR(aDatei[F_NAME],10), aDatei[F_SIZE]),;
    nTotal += aDatei[F_SIZE] })
?
? "Gesamtgr..e :", nTotal
```

. Das n.chste Beispiel benutzt AEVAL(), um eine Liste mit Dateinamen zu erzeugen, welche aus einem mehrdimensionalen Array entnommen werden:

```
#include "DIRECTRY.CH"
LOCAL aDateien := DIRECTORY("*.dbf"), aNamen := {}
AEVAL(aDateien,{ | aDatei AADD( aNamen, aDatei[F_NAME]) } )
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SAVE Sichert Variablen in einer .MEM-Datei

Syntax

SAVE TO <xcMemDatei> [ALL [LIKE | EXCEPT <VariablenSpezifikation>]]

Argumente

<xcMemDatei> gibt die auf der Platte zu speichernde .MEM-Datei an. Der Name kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Falls keine andere Erweiterung angegeben ist, wird die Standard-Erweiterung .MEM angenommen.

ALL [LIKE EXCEPT <VariablenSpezifikation>] definiert eine Gruppe von sichtbaren PRIVATE- und PUBLIC-Speichervariablen, die in der .MEM-Datei gesichert (LIKE) bzw. nicht gesichert werden soll (EXCEPT). Dabei gibt <VariablenSpezifikation> die entsprechenden Speichervariablen an. Wildcards (* und ?) sind hierbei zugelassen.

Anwendung

SAVE kopiert PUBLIC- und PRIVATE-Speichervariablen, die in der laufenden Prozedur oder UDF sichtbar sind, in eine .MEM-Datei (Speichervariablen-Datei). Arrays, LOCAL- und STATIC-Variablen k.nnen auf diese Weise jedoch nicht gesichert werden. Die Speicherung der Variablen erfolgt ohne die jeweiligen G.ltigkeitsbereiche. Variablen, die durch explizite PRIVATE- oder LOCAL-Deklarationen versteckt worden sind, werden ebenfalls nicht gesichert.

Die ALL LIKE-Klausel bewirkt, da. alle, die <VariablenSpezifikation> erf.llenden Variablen mit SAVE gesichert werden. Im Gegensatz dazu schlie.t die ALL EXCEPT-Klausel nur die Variablen in die Sicherung ein, welche die <VariablenSpezifikation> nicht erf.llen.

Die <VariablenSpezifikation> darf sog. Wildcards (die Zeichen * und ?) enthalten. Das *-Zeichen bedeutet dabei, das alle nachfolgenden Zeichen beliebig sein k.nnen und nur die dem *-Zeichen vorangehenden Zeichen .bereinstimmen m.ssen. Das ?-Zeichen bedeutet, da. dieses einzelne Zeichen an dieser bestimmten Position beliebig sein darf.

Beispiele

. In diesem Beispiel werden alle PRIVATE- und PUBLIC-Variablen in Temp.Mem gesichert:

```
PRIVATE cEins := "1"
SAVE ALL TO Temp
```

. In diesem Beispiel werden alle sichtbaren PRIVATE- und PUBLIC-Variablen, deren Name mit c beginnt, in Temp.Mem gesichert:

```
SAVE ALL LIKE c* TO Temp
```

. In diesem Beispiel werden alle sichtbaren PRIVATE- und PUBLIC-Variablen, deren Name nicht mit c beginnt, in Temp.Mem gesichert:

```
SAVE ALL EXCEPT c* TO Temp
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SAVE SCREEN* Sichert den aktuellen Bildschirminhalt in einem Puffer oder einer Variablen

---

Syntax

SAVE SCREEN [TO <idVar>]

Argumente

TO <idVar> gibt die Variable an, welcher der aktuelle Bildschirminhalt als Zeichenwert zugewiesen werden soll. Wenn diese Variable nicht existiert oder nicht sichtbar ist, wird eine PRIVATE-Variable angelegt.

Anwendung

SAVE SCREEN ist der synonyme Befehl f.r die Funktion SAVESCREEN(), wenn diese den Bildschirm von 0,0 bis MAXROW(), MAXCOL() in einen vorgegebenen Bildschirmpuffer oder die angegebene Variable sichert. Die Variable kann zu einer beliebigen Speicherklasse (z.B. FIELD, LOCAL, STATIC, Array) geh.ren. LOCAL-, Array und STATIC-Variablen k.nnen aber nicht mit SAVE in einer .MEM-Datei gespeichert werden, um so verschiedene Bildschirminhalte auf Platte zu sichern.

SAVE SCREEN wird in Kombination mit RESTORE SCREEN eingesetzt, um den zeitaufwendigen Wiederaufbau eines vorher kurzzeitig .berschriebenen Bildschirmes zu vermeiden. Sie k.nnen mehrere verschiedene Bildschirminhalte sichern, indem Sie diese Inhalte Variablen mit unterschiedlichen Namen zuweisen.

SAVE SCREEN ist ein Kompatibilit.ts-Befehl und wird daher nicht zum Einsatz in neuen Applikationen empfohlen. Er wird durch die Funktion SAVESCREEN() ersetzt, die neben kompletten Bildschirminhalten auch Teile davon sichern kann.

Achtung

SAVE SCREEN, RESTORE SCREEN, SAVESCREEN() und RESTSCREEN() werden nur bei Verwendung des vorgegebenen Bildschirmtreibers (IBM-PC, memory mapped) unterst.tzt. Andere Bildschirmtreiber unterst.tzen diese M.glichkeiten u.U. nicht.

Beispiele

. Dieses Programmfragment zeigt die M.glichkeit der Speicherung von Bildschirminhalten in STATIC-Variablen:

```
STATIC aBildschirme[10]
SAVE SCREEN TO aBildschirme[1]
//
<Anweisungen>...
//
RESTORE SCREEN FROM acBildschirme[1]
```

. Das folgende Beispiel demonstriert den Einsatz von einer Datenbank zur Speicherung und Wiederherstellung von Bildschirminhalten:

```
USE Masken INDEX Name NEW
APPEND BLANK
Masken->Name := "Maske001"              // Maskenname speichern
SAVE SCREEN TO Masken->Inhalt           // Bildschirminhalt sichern
//
<Anweisungen>...
//
SEEK "Maske001"                         // Suche Namen
RESTORE SCREEN FROM Masken->Inhalt      // Wiederaufbau des Bildschirmes
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SEEK Durchsucht den Index nach einem angegebenen Schl.sselausdruck

---

Syntax

SEEK <SuchAusdruck>

Argumente

<SuchAusdruck> ist ein Ausdruck, der mit einem Indexschl.ssel .bereinstimmen soll.

Anwendung

SEEK ist ein Datenbankbefehl, der den aktiven kontrollierenden Index ab dem ersten Schl.ssel durchsucht, bis ein passender Eintrag oder ein Eintrag gefunden wird, der gr..er als der <SuchAusdruck> ist. Bei erfolgreicher Suche wird der Datensatzzeiger auf den gefundenen Datensatz gesetzt.
Abh.ngig von der Einstellung von SOFTSEEK ergeben sich nach einem SEEK unterschiedliche Ergebnisse. Ist SOFTSEEK auf OFF gestellt (Vorgabe), wird der Datensatzzeiger bei erfolgloser Suche auf LASTREC()+1 gesetzt, EOF() gibt true (.T.) und FOUND() false (.F.) zur.ck. Ist SOFTSEEK auf ON gestellt, wird der Datensatzzeiger bei erfolgloser Suche auf den ersten Satz gesetzt, dessen Schl.sselwert gr..er als das Suchargument ist. FOUND() gibt false (.F.) zur.ck. EOF() gibt in diesem Fall nur true (.T.) zur.ck, wenn es im Index keine Schl.sselwerte gibt, die gr..er als das Suchargument sind.

SET EXACT beeinflu.t den Suchvorgang mit SEEK nicht.

Beispiele

. Dieses Beispiel zeigt m.gliche Ergebnisse von SEEK:

```
USE Verkauf INDEX Verkauf NEW
FIND ("100")
? FOUND(), EOF(), RECNO()   // Ergebnis: .T. .F. 1
```

. Die folgende UDF f.hrt eine exakte Suche nach einem Index mit einem Zeichenfeld als Schl.ssel durch:

```
USE Rechnung INDEX Rechnung NEW
IF SucheGenau("10001")
  ? "Rechnung ist vorhanden"
ELSE
  ? "Rechnung ist nicht vorhanden"
ENDIF
RETURN
FUNCTION SucheGenau(cSuchen)
  SEEK PADR(cSuchen, LEN(&(INDEXKEY(0))))
RETURN (FOUND())
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SELECT Wechselt den aktuellen Arbeitsbereich

Syntax

SELECT <xnArbeitsBereich> | <idAlias>

Argumente

<xnArbeitsBereich> ist eine Zahl von 0 bis 250, die den Arbeitsbereich angibt. Dieses Argument ist ein erweiterter Ausdruck und kann entweder als numerisches Literal oder als eingeklammerter Ausdruck angegeben werden.

<idAlias> ist der Name eines existierenden Arbeitsbereiches, der selektiert werden soll, falls dort eine Datenbank ge.ffnet ist.

Anwendung

SELECT ist ein Datenbankbefehl, mit dem Sie den aktuellen Arbeitsbereich wechseln k.nnen. Clipper unterst.tzt 250 Arbeitsbereiche, wobei jeder Arbeitsbereich einen logischen Handle f.r eine ge.ffnete Datenbank samt aller zugeh.rigen Attribute darstellt. Bezugnahmen auf Arbeitsbereiche k.nnen .ber SELECT unter Angabe einer Zahl oder des Alias-Namens erfolgen. Der Alias-Name eines Arbeitsbereiches wird automatisch bei .ffnung einer Datenbank zugewiesen (dann ist es der Dateiname ohne Erweiterung), oder er wird .ber die Alias-Klausel bei USE spezifiziert.

Der Arbeitsbereich Null bezieht sich auf den ersten freien Arbeitsbereich. Sie k.nnen daher eine Datenbank mit SELECT 0 und USE <xcDatenbank> im jeweils n.chsten, freien Bereich .ffnen.

Hinweise

. Alias-Ausdr.cke: Eine wesentlich funktionellere Methode zur Selektierung von neuen Arbeitsbereichen besteht in der Verwendung von Alias-Ausdr.cken. Anstatt mit mehreren SELECT-Befehlen zu operieren, k.nnen Sie dem auszuf.hrenden Ausdruck einfach einen entsprechenden eingeklammerten Alias-Verweis voranstellen. Als Beispiel sei hier die Abfrage von EOF() in einem nicht selektierten Arbeitsbereich genannt. Auf herk.mmliche Art sieht die Abfrage so aus:

```
SELECT Andere
? EOF()
SELECT Ursprung
```

. Mit dem Einsatz eines Alias-Ausdrucks sieht der Vorgang erheblich k.rzer aus:

```
? Andere->(EOF())
```

. USE...NEW: Wie oben schon erw.hnt, k.nnen Sie eine Datenbank in einem neuen Arbeitsbereich mit SELECT 0 und USE <xcDatenbank> .ffnen. Clipper 5.0 stellt nun einen einfacheren Weg zur Verf.gung:

```
USE <xcDatenbank> NEW
```

Beispiele

. Dieses Beispiel zeigt, wie mehrere Datenbanken nach Selektierung jedes einzelnen Arbeitsbereiches .ber eine neue Zahl mit USE ge.ffnet werden:

```
SELECT 1
USE Kunden
SELECT 2
USE Rechnung
SELECT 3
USE Posten
SELECT Kunden
```

. Die bessere Methode zur .ffnung von (mehreren) Dateien ist die Verwendung von USE...NEW:

```
USE Kunden NEW
USE Rechnung NEW
USE Posten NEW
SELECT Kunden
```

. Dieses Programmfragment zeigt, wie der Name des aktuellen Arbeitsbereiches in einer Variablen gesichert wird. Danach wird ein neuer Arbeitsbereich selektiert, die gew.nschten Anweisungen werden ausgef.hrt, und der alte Arbeitsbereich wird wieder gew.hlt:

```
nAlterBereich := SELECT()
USE NeuDatei NEW
//
<Anweisungen>...
//
SELECT (AlterBereich)
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET ALTERNATE Gibt die Ergebnisse von Konsolen-Befehlen zus.tzlich in eine Textdatei aus

---

Syntax

SET ALTERNATE TO [<xcDatei>]
SET ALTERNATE on | OFF | <xlSchalter>

Argumente

<xcDatei> ist der Name einer Standard-ASCII-Datei, die als Zieldatei f.r Ausgaben dient. Der Name kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Falls keine andere Erweiterung angegeben ist, wird die Standard-Erweiterung .TXT hinzugef.gt. Optional kann ein Laufwerk und/oder Pfad angegeben werden. Falls eine Datei gleichen Namens bereits existiert, wird sie ohne Vorwarnung .berschrieben.

ON aktiviert die Ausgabe von Konsolen-Befehlen in die ge.ffnete Textdatei.

OFF deaktiviert die Ausgabe von Konsolen-Befehlen in die ge.ffnete Textdatei, ohne allerdings die Datei zu schlie.en.

<xlSchalter> ist ein logischer Ausdruck, der eingeklammert werden mu.. Dabei hat true (.T.) die gleiche Bedeutung wie ON und false (.F.) entspricht OFF.

Anwendung

SET ALTERNATE ist ein Konsolen-Befehl, der die Ausgabe von Konsolen-Befehlen in eine Textdatei schreibt. Unter Konsolen-Befehlen werden Befehle wie LIST, REPORT FORM, LABEL FORM oder ? verstanden, die allesamt Bildschirmausgaben ohne Angabe einer Zeilen- oder Spaltennummer vornehmen. Die meisten dieser Befehle k.nnen mit einer TO FILE-Klausel aufgerufen werden, wobei dies die gleiche Funktion wie SET ALTERNATE erf.llt. SET ALTERNATE kann nicht f.r Full-screen-Befehle wie @...SAY verwendet werden. Die Ausgaben dieser Befehle k.nnen nur .ber eine Kombination von SET DEVICE TO PRINTER und SET PRINTER TO <xcDatei> in eine Datei geschrieben werden.

SET ALTERNATE hat zwei verschiedene Argumentformen. TO <xcDatei> erzeugt eine DOS-Textdatei mit der Standard-Erweiterung .TXT, wobei eine bereits vorhandene, gleichnamige Datei .berschrieben wird. Eine Alternate-Datei ist nicht einem bestimmten Arbeitsbereich zugeordnet. Es k.nnen nicht mehrere Alternate-Dateien gleichzeitig ge.ffnet sein. Sie k.nnen eine Alternate-Datei mit CLOSE ALTERNATE, CLOSE ALL oder SET ALTERNATE TO ohne Argument schlie.en.

Die Argumentform on|OFF kontrolliert den Schreibvorgang der Konsolen-Ausgabe in die aktuelle Alternate-Datei. Der Schreibproze. beginnt durch Angabe von SET ALTERNATE ON. Er kann durch SET ALTERNATE OFF unterbrochen werden, ohne die Ausgabedatei zu schlie.en.

Beispiele

Dieses Beispiel erzeugt eine Alternate-Datei und schreibt die Ergebnisse der Ausgaben mit ? f.r jeden Datensatz in der Datenbank in diese Ausgabedatei:

```
SET ALTERNATE TO ListDat
SET ALTERNATE ON
USE Kunden NEW
DO WHILE .NOT. EOF()
  ? Kunden->Nachname, Kunden->Ort
  SKIP
ENDDO
SET ALTERNATE OFF
CLOSE ALTERNATE
CLOSE Kunden
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET BELL Schaltet den Signalton bei Full-screen-Operationen ein/aus

---

Syntax

SET BELL on | OFF | <xlSchalter>

Argumente

ON aktiviert den Signalton.

OFF deaktiviert den Signalton.

<xlSchalter> ist ein logischer Ausdruck, der eingeklammert werden mu..
Dabei hat true (.T.) die gleiche Bedeutung wie ON und false (.F.)
entspricht OFF.

Anwendung

SET BELL ist ein Environment-Befehl, der den Signalton ein- und
ausschaltet. Falls der Signalton eingeschaltet ist, wird er in folgenden
Situationen ausgegeben:

. Der Benutzer gibt an der letzten Position eines GET-Eingabefeldes ein
Zeichen ein.

. Der Benutzer versucht, in ein GET-Eingabefeld ung.ltige Daten
einzugeben. Die jeweils g.ltigen Daten werden durch den Datentyp der
GET-Variablen, die PICTURE-Formatierungsschablone und die RANGE-Klausel
bestimmt. Ein Versto. gegen die VALID-Bedingung f.hrt jedoch in keinem
Fall zu einer Ausgabe des Signaltons.

Um den Signalton in anderen Situationen explizit auszugeben, k.nnen Sie
?? CHR(7) oder die TONE()-Funktion verwenden. Der Einsatz von TONE() ist
sicherlich vorzuziehen, da Sie hierbei die Tonh.he und -dauer ver.ndern
k.nnen.

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET CENTURY .ndert das Datumsformat, indem die Jahrhundertziffern ein- oder ausgeschlossen werden

---

Syntax

SET CENTURY on | OFF | <xlSchalter>

Argumente

ON aktiviert die Ein- und Ausgabe der Jahrhundertziffern in einem Datum.

OFF deaktiviert die Ein- und Ausgabe der Jahrhundertziffern in einem Datum.

<xlSchalter> ist ein logischer Ausdruck, der eingeklammert werden mu.. Dabei hat true (.T.) die gleiche Bedeutung wie ON und false (.F.) entspricht OFF.

Anwendung

SET CENTURY ON bewirkt, da. das Datumsformat eine vierstellige Jahreszahl enth.lt. In diesem Fall wird jedes Datum mit einer vierstelligen Jahreszahl ausgegeben, und Eingaben k.nnen entsprechend f.r jedes Jahrhundert erfolgen.

SET CENTURY OFF bewirkt, da. das Datumsformat eine zweistellige Jahreszahl enth.lt. In diesem Fall wird jedes Datum mit einer zweistelligen Jahreszahl ausgegeben, und Eingaben f.r Jahrhunderte k.nnen nicht erfolgen.

Der SET CENTURY-Befehl betrifft nur Ein- und Ausgaben. Berechungen mit Datumsfeldern schlie.en das Jahrhundert immer mit ein, unabh.ngig vom Datumsformat.

Clipper unterst.tzt Daten vom 01.01.0100 bis zum 31.12.2999.

Beispiele

Das folgende Beispiel zeigt die Ergebnisse von SET CENTURY-Befehlen:

```
SET CENTURY OFF
? DATE()         // Ergebnis: 31.12.89
SET CENTURY ON
? DATE()         // Ergebnis: 31.12.1989
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET COLOR* Definiert die Bildschirmfarben

Syntax

```
SET COLOR | COLOUR TO
  [[<standard>]
   [,<enhanced>]
   [,<border>]
   [,<background>]
   [,<unselected>]] | (<cFarbenString>)
```

Argumente

<standard legt die Farbe f.r die Bildschirmausgaben von allen Konsolen- sowie Full-screen-Befehlen und -Funktionen fest. Dazu geh.ren z.B. @...PROMPT, @...SAY und ? im Bereich der Befehle und ACHOICE(), DBEDIT() und MEMOEDIT() im Bereich der Funktionen.

<enhanced> legt die Farbe f.r Hervorhebungen und Markierungen fest. Dazu geh.ren z.B. GETs mit SET INTENSITY ON, MENU TO, DBEDIT() und die Men.auswahl-Markierung von ACHOICE().

<border> legt die Farbe f.r den nicht beschreibbaren Bereich rund um den physischen Bildschirm fest.

<background> wird zur Zeit von keinem Computertyp unterst.tzt, f.r den Nantucket Treiber liefert. Die Angabe f.r <background> ist nur aus Kompatibilit.tsgr.nden verf.gbar.

<unselected> gibt ein Farbenpaar an, das das augenblicklich aktuelle GET-Eingabefeld der Farbe <enhanced> anzeigt, w.hrend alle anderen GETs in dieser <unselected>-Farbe angezeigt werden.

<cFarbenString> ist ein eingeklammerter Zeichenkettenausdruck, welcher die Farbeinstellungen liefert. Hiermit k.nnen Sie die Farbeinstellungen auch als Ausdruck anstatt eines String-Literals oder einer Makro-Variablen angeben.

Die Angabe SET COLOR TO ohne Argument stellt die Vorgabefarben W/N, N/W, N, N, N/W wieder her.

Anwendung

SET COLOR ist der synonyme Befehl f.r die Funktion SETCOLOR(), welche die Farben f.r die nachfolgenden Bildschirmausgaben spezifiziert. Jeder SET COLOR-Befehl gibt eine Liste mit bis zu f.nf Farbeinstellungen f.r die entsprechenden Typen von Bildschirmausgaben an. Jede Farbeinstellung besteht aus einer Angabe f.r die Vordergrundfarbe, dem Trennzeichen / und der Angabe f.r die Hintergrundfarbe. Dabei bezieht sich die Vordergrundfarbe auf die einzelnen dargestellten Zeichen, w.hrend die Hintergrundfarbe den farblichen Untergrund f.r das einzelne Zeichen festlegt. Leerzeichen und nicht darstellbare (Steuer-)Zeichen werden immer in der Hintergrundfarbe ausgegeben.

Die Einstellung des Vordergrundes kann neben der Farbe auch noch ein Attribut enthalten, n.mlich intensiv und blinkend. Bei einem Monochrom-Monitor bewirkt die intensive Darstellung, da. der Text heller als normal dargestellt wird. Bei einem Farb-Monitor ver.ndert das Intensit.tsattribut die Farbs.ttigung und somit die Farbe als solches. Z.B. stellt "N" den Vordergrundtext schwarz dar, w.hrend "N+" den Text grau ausgibt. Damit ist auch bereits das Zeichen f.r das Intensit.tsattribut vorgestellt worden, es ist das Pluszeichen (+). Bei Angabe des Blinkattributes (das Zeichen *) wird der Vordergrundtext in regelm..igen Abst.nden ein- und ausgeschaltet. Ein Attributzeichen kann an jeder Stelle der Farbeinstellung stehen, es wirkt aber immer nur auf die Vordergrundfarbe, unabh.ngig von seiner Position.

Jede Farbe kann als Zahl oder Buchstabe angesprochen werden, eine Vermischung beider Darstellungen ist allerdings nicht m.glich. Zahlen sind nur aus Kompatibilit.tsgr.nden vorhanden und daher nicht zum

Einsatz in neuen Applikationen empfohlen.

Alle Einstellungen sind optional. Falls eine Farbeinstellung .bersprungen wird, bleiben ihre vorherigen Werte g.ltig. Wenn Sie eine Vordergrund- oder Hintergrundfarbe auslassen, wird stattdessen schwarz als Farbe genommen.

Clipper unterst.tzt die folgenden Farben:

Tabelle 4-13: Farbtabelle

Farbe	Buchstabe	Zahl	Monochrom-Darstellung
Schwarz	N, Leerzeichen	0	Schwarz
Blau	B	1	Unterstrichen
Gr.n	G	2	Wei.
T.rkis	BG	3	Wei.
Rot	R	4	Wei.
Magenta	RB	5	Wei.
Braun	GR	6	Wei.
Hellgrau	W	7	Wei.
Dunkelgrau	N+	8	Schwarz
Hellblau	B+	9	Intensiv Unterstrichen
Hellgr.n	G+	10	Intensiv Wei.
Hellt.rkis	BG+	11	Intensiv Wei.
Hellrot	R+	12	Intensiv Wei.
Hell-Magenta	RB+	13	Intensiv Wei.
Gelb	GR+	14	Intensiv Wei.
Wei.	W+	15	Intensiv Wei.
Schwarz	U		Unterstrichen
Invers	I		Invers
Transparant	X		Transparent

SET COLOR ist ein Kompatibilit.ts-Befehl und wird daher nicht zum Einsatz in neuen Applikationen empfohlen. Er wird durch die Funktion SETCOLOR() ersetzt, die neben der .nderung von Farben auch die aktuelle Einstellung zur.ckgeben kann.

Hinweise

. Monochrom-Monitore: Clipper unterst.tzt aber die monochromen Attribute invers und unterstrichen.

. Bildschirmtreiber: Die Zahlenangaben bei SET COLOR TO werden u.U. nicht unterst.tzt, wenn Sie einen anderen als den vorgegebenen Bildschirmtreiber verwenden.

Beispiele

. Das folgende Beispiel demonstriert den Einsatz von unselected, um das aktuelle GET-Eingabefeld in den Farben rot auf hellgrau und den Rest in schwarz auf hellgrau darzustellen:

```
cFarbe = "W/N,R/W,,,N/W"
SET COLOR TO (cFarbe)
cEins := cZwei := SPACE(10)
@ 1, 1 SAY "Eingabe eins:" GET cEins
@ 2, 1 SAY "Eingabe zwei:" GET cZwei
READ
```

. Dieses Beispiel zeigt, wie eine UDF zur Passwortabfrage verwendet werden kann, wobei die Eingabe nicht sichtbar ist, weil die <enhanced>-Einstellung "X" (transparent) ist:

```
IF .NOT. DialogPassWort(12, 13, "W+/N", "DINGDONG", 3)
  ? "Ihr Passwort ist leider falsch!"
  QUIT
ENDIF

FUNCTION DialogPassWort(nZeile,nSpalte,cStandard,cPasswort,nVersuche)
  LOCAL nZaehler := 1, cFarbe := SETCOLOR()
  SET COLOR TO (cStandard + ", X")  // Tranparente Eingabe
```

```
  //
  DO WHILE nZaehler < nVersuche
    cBenutzerEingabe = SPACE(6)
    @ nZeile, nSpalte SAY  "Passwort: " GET cBenutzerEingabe
    READ
    //
    IF LASTKEY() == 27
      SET COLOR TO (cFarbe)
      RETURN .F.
    ELSEIF cBenutzerEingabe == cPasswort
      SET COLOR TO (cFarbe)
      RETURN .T.
    ELSE
      nZaehler++
    ENDIF
  ENDDO
  //
  SET COLOR TO (cFarbe)
RETURN .F.
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET CONFIRM Schaltet die Notwendigkeit einer Abschlu.taste zur Beendigung von GETs ein/aus

---

Syntax

SET CONFIRM on | OFF | <xlSchalter>

Argumente

ON bewirkt, da. der Benutzer zum Verlassen eines GET-Feldes eine Abschlu.taste dr.cken mu..

OFF bewirkt, da. der Benutzer zum Verlassen eines GET-Feldes keine Abschlu.taste dr.cken mu., sondern dieses bereits bei Erreichen des Eingabefeld-Endes verl..t.

<xlSchalter> ist ein logischer Ausdruck, der eingeklammert werden mu.. Dabei hat true (.T.) die gleiche Bedeutung wie ON und false (.F.) entspricht OFF.

Anwendung

SET CONFIRM bestimmt, ob zum Verlassen eines GET-Eingabefeldes das Dr.cken einer Abschlu.taste notwendig ist oder nicht. Steht SET CONFIRM auf OFF, gelangt der Cursor direkt in das n.chste Eingabefeld, wenn der Benutzer .ber das Ende des aktuellen GET-Feldes hinausschreibt. Gibt es kein weiteres Eingabefeld, wird READ beendet. Wenn jedoch SET CONFIRM auf ON steht, mu. in jedem Fall eine Abschlu.taste gedr.ckt werden, damit ein GET verlassen werden kann.

In allen genannten F.llen wird nach Verlassen des aktuellen GET-Feldes eine evtl. vorhandene RANGE- oder VALID-Klausel ausgewertet, au.er es wurde [Esc] gedr.ckt.

In der Beschreibung von @...GET finden Sie eine detaillierte Darstellung .ber das Verhalten von GET-Feldern.

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET CONSOLE Schaltet die Ausgabe von Konsolen-Befehlen auf den Bildschirm ein/aus

---

Syntax

SET CONSOLE ON | off | <xlSchalter>

Argumente

ON aktiviert die Ausgabe von Konsolen-Befehlen auf dem Bildschirm.

OFF unterdr.ckt die Ausgabe von Konsolen-Befehlen auf dem Bildschirm.

<xlSchalter> ist ein logischer Ausdruck, der eingeklammert werden mu.. Dabei hat true (.T.) die gleiche Bedeutung wie ON und false (.F.) entspricht OFF.

Anwendung

SET CONSOLE bestimmt, ob die Ausgabe von Konsolen-Befehlen auch an den Bildschirm gesendet werden. Konsolen-Befehle nehmen Bildschirmausgaben ohne Angabe einer Zeilen- oder Spaltennummer vor. Zus.tzlich zur Bildschirmausgabe k.nnen Konsolen-Befehle auch gleichzeitig auf dem Drucker und/oder in eine DOS-Textdatei ausgegeben werden. Die Ausgabe auf dem Drucker wird mit der bei den meisten Konsolen-Befehlen vorhandenen Klausel TO PRINTER oder dem Befehl SET PRINTER ON aktiviert. Ausgaben in eine Datei verwenden die TO FILE-Klausel, SET ALTERNATE oder SET PRINTER TO.

Ist SET CONSOLE auf ON eingestellt, werden Konsolen-Ausgaben auf dem Bildschirm angezeigt. Bei Einstellung auf OFF erfolgt keine Darstellung auf dem Bildschirm, aber Ausgaben zum Drucker oder in eine Datei bleiben davon unber.hrt. Dadurch k.nnen sinnvollerweise (in der Regel l.ngere) Ausgaben mit REPORT FORM oder LABEL FORM .ber den Drucker erfolgen, ohne da. sie auf dem Bildschirm sichtbar sind.

Hinweise

. Tastatureingaben: Bei Konsolen-Befehlen, die Eingaben erwarten (z.B. ACCEPT, INPUT oder WAIT), betrifft SET CONSOLE sowohl die Ausgabe der PROMPTs als auch die Eingabebereiche. Folgerichtig bedeutet die Angabe von SET CONSOLE OFF vor einem solchen Befehl, da. f.r den Benutzer sowohl die Eingabeanforderung als auch die Eingabe selbst unsichtbar bleiben.

. Full-screen-Befehle: Full-screen-Befehle wie @...SAY nehmen ihre Bildschirmausgaben ohne Ber.cksichtigung von SET CONSOLE vor. F.r diese Art von Ausgabebefehlen mu. SET DEVICE verwendet werden, um als Ausgabeeinheit den Bildschirm oder den Drucker zu bestimmen. SET PRINTER TO leitet die Ausgabe in eine Datei.

Beispiele

Das folgende Beispiel gibt einen Bericht mit REPORT FORM aus und unterdr.ckt dabei die Ausgabe auf den Bildschirm:

```
USE Verkauf NEW
SET CONSOLE OFF
REPORT FORM Verkauf TO PRINTER
SET CONSOLE ON
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET CURSOR Schaltet den Bildschirm-Cursor ein/aus

Syntax

SET CURSOR on | OFF | <xlSchalter>

Argumente

ON aktiviert die Cursoranzeige.

OFF deaktiviert die Cursoranzeige.

<xlSchalter> ist ein logischer Ausdruck, der eingeklammert werden mu..
Dabei hat true (.T.) die gleiche Bedeutung wie ON und false (.F.)
entspricht OFF.

Anwendung

SET CURSOR schaltet die Anzeige des Bildschirm-Cursors ein bzw. aus.
Wenn der Cursor ausgeschaltet ist, bleiben Tastatureingaben und
Bildschirmausgaben davon unber.hrt. Der Cursor wird lediglich versteckt,
und eine Dateneingabe kann ohne den sichtbaren Cursor weiterhin
vorgenommen werden. ROW() und COL() werden normal aktualisiert, als ob
der Cursor sichtbar w.re.

Dieser Befehl dient vor allem dazu, den Cursor bei einem
Bildschirmaufbau zu unterdr.cken. Im besten Fall ist der Cursor in einem
Anwendungsprogramm nur dann sichtbar, wenn der Benutzer Eingaben
vornehmen mu. (z.B. bei GETs oder MEMOEDIT()).

Beispiele

Das folgende Beispiel zeigt eine typische Verwendung von SET CURSOR:

```
LOCAL lAntwort := .F.
@ 24, 0
@ 24, 15 SAY "Wollen Sie das Programm beenden? (J/N)";
  GET lAntwort PICTURE "Y"
SET CURSOR ON
READ
SET CURSOR OFF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET DATE Bestimmt das Datumsformat f.r Ein- und Ausgaben

Syntax

```
SET DATE [TO]
  AMERICAN | ansi  | british  | french  | german  | italian  | japan  | usa
SET DATE FORMAT [TO] <xcDatumsFormat>
```

Argumente

<xcDatumsFormat> ist ein Zeichenausdruck, der das Datumsformat angibt, wenn die Klausel FORMAT benutzt worden ist. <xcDatumsFormat> mu. eine Zeichenkette von 12 oder weniger Zeichen ergeben.

Falls Sie <xcDatumsFormat> angegeben haben, wird der Ausdruck auf die g.ltige Verwendung und Positionierung der Buchstaben f.r die Ziffern f.r Tag, Monat und Jahr hin analysiert. Die Position der Zahlen f.r Tag, Monat und Jahr wird durch Durchsuchen von <xcDatumsFormat> nach ein- oder mehrmaligem Vorkommen von d, m und y ermittelt.
Alle anderen Zeichen werden eins zu eins an die angegebene Position im Ausgabewert kopiert.

Anwendung

SET DATE ist ein Environment-Befehl, der das Ausgabeformat f.r Datumswerte festlegt. SET DATE ist eine globale Einstellung, die das Verhalten aller Datumswerte im gesamten Programm betrifft. Die .nderung der Datumsformatierung erleichtert Ihnen die Portierung Ihrer Applikationen in fremde Sprachen.

Tabelle 4-14: Formate von SET DATE

Einstellung	Format
AMERICAN	mm/dd/yy
ANSI	yy.mm.dd
BRITISH	dd/mm/yy
FRENCH	dd/mm/yy
GERMAN	dd.mm.yy
ITALIAN	dd-mm-yy
JAPAN	yy/mm/dd
USA	mm-dd-yy

Beispiele

. Das folgende Beispiel demonstriert den Einsatz der FORMAT-Klausel, um das Datumsformat direkt anzugeben:

```
SET DATE FORMAT "dd:mm:yyyy"
```

. Dieses Beispiel zeigt, wie das Datumsformat f.r die Programmausf.hrung durch .bergabe einer DOS-Umgebungsvariablen konfiguriert werden kann, deren Inhalt mit GETENV() ermittelt wird. Diesen so ermittelten Wert verarbeitet SET DATE.

```
C>SET CLIP_DATE=dd/mm/yy
```

Im Konfigurationsteil des Anwendungsprogrammes wird das Datumsformat dann wie folgt eingestellt:

```
FUNCTION AppKonfig
  SET DATE FORMAT TO GETENV("CLIP_DATE")
RETURN NIL
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET DECIMALS Stellt die Anzahl der auszugebenden Dezimalstellen ein

---

Syntax

SET DECIMALS TO [<nDezimalstellen>]

Argumente

<nDezimalstellen> gibt die Anzahl der Dezimalstellen bei Ausgaben an. Die Vorgabe sind zwei Dezimalstellen.

SET DECIMALS TO ohne Argument ist gleichbedeutend mit SET DECIMALS TO 0.

Anwendung

SET DECIMALS bestimmt die Anzahl der Dezimalstellen bei Ausgaben der Ergebnisse von numerischen Funktionen und Berechnungen. Dabei h.ngt SET DECIMALS direkt mit SET FIXED zusammen. Ist SET FIXED auf OFF gestellt, werden nur die jeweils ben.tigten Dezimalstellen bei EXP(), LOG(), SQRT() und Divisionen ausgegeben.
Ist SET FIXED auf ON eingestellt, dann werden alle numerischen Werte mit genau der bei SET DECIMALS festgelegten Dezimalstellenanzahl angezeigt. Sowohl der SET DECIMALS- als auch der SET FIXED-Befehl ber.hren nicht die Rechengenauigkeit, sondern nur das Ausgabeformat.

Eine noch pr.zisere Einstellung des numerischen Ausgabeformates ist mit der PICTURE-Klausel von @...SAY und @...GET oder der Funktion TRANSFORM() m.glich.

Beispiele

Das folgende Beispiel zeigt verschiedene Ergebnisse des SET DECIMALS-Befehles:

```
SET FIXED ON
SET DECIMALS TO 2    // Vorgabeeinstellung
? 2/4                // Ergebnis: 0.50
? 1/3                // Ergebnis: 0.33
SET DECIMALS TO 4
? 2/4                // Ergebnis: 0.5000
? 1/3                // Ergebnis: 0.3333
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET DEFAULT Legt das Vorgabelaufwerk und -verzeichnis f.r Clipper fest

---

Syntax

SET DEFAULT TO [<xcPfadSpezifikation>]

Argumente

<xcPfadSpezifikation> kennzeichnet das angegebene Laufwerk und Verzeichnis als Vorgabepfad und kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Nach dem Laufwerksbuchstaben mu. ein Doppelpunkt angegeben werden, wenn dahinter ein Verzeichnisname folgt. SET DEFAULT ohne Argument macht das aktuelle DOS-Laufwerk und -Verzeichnis zum Vorgabepfad.

Anwendung

SET DEFAULT bestimmt das Laufwerk und Verzeichnis, in dem das Applikationsprogramm Dateien anlegt und speichert, mit der Ausnahme von tempor.ren und mit den Low-level-Dateifunktionen erzeugten Dateien.

SET DEFAULT ver.ndert nicht das DOS-Laufwerk und -Verzeichnis. Beim Zugriff auf Dateien wird zuerst im SET DEFAULT-Pfad gesucht. Weitere Suchpfade k.nnen Sie mit SET PATH bestimmen.

Hinweise

. Startvorgabe: Nach dem Start eines Clipper-Programmes ist das aktuelle DOS-Laufwerk und -Verzeichnis der SET DEFAULT-Pfad. Dies k.nnen Sie w.hrend der Programmausf.hrung mit SET DEFAULT .ndern.

. Aufruf von externen Programmen: Bei der Ausf.hrung des RUN-Befehles wird auf das aktuelle DOS-Laufwerk und -Verzeichnis zugegriffen.

Beispiele

Das folgende Beispiel zeigt eine typische Anwendung von SET DEFAULT:

```
SET PATH TO
? FILE("Verkauf.DBF")              // Ergebnis: .F.
//
SET DEFAULT TO C:\CLIPPER\DATEIEN
? FILE("Verkauf.DBF")              // Ergebnis: .T.
//
SET DEFAULT TO C:    // Anderes Vorgabelaufwerk
SET DEFAULT TO \     // Wechsel in das Hauptverzeichnis
SET DEFAULT TO ..    // Wechsel in das n.chsth.here Verzeichnis
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET DELETED Schaltet den Filter f.r als gel.scht markierte Datens.tze ein/aus

---

Syntax

SET DELETED on | OFF | <xlSchalter>

Argumente

ON ignoriert als gel.scht markierte Datens.tze.

OFF ber.cksichtigt als gel.scht markierte Datens.tze.

<xlSchalter> ist ein logischer Ausdruck, der eingeklammert werden mu.. Dabei hat true (.T.) die gleiche Bedeutung wie ON und false (.F.) entspricht OFF.

Anwendung

SET DELETED schaltet die automatische Filterung von zur L.schung markierten Datens.tzen in allen Arbeitsbereichen ein bzw. aus. Ist SET DELETED auf ON gestellt, ignorieren die meisten Befehle gel.schte Datens.tze. Wenn Sie jedoch einen Datensatz .ber die Satznummer ansprechen (GOTO oder alle Befehle, die als <Wirkungsbereich> RECORD unterst.tzen), wird dieser Datensatz nicht ignoriert, auch wenn er als gel.scht markiert ist. SET DELETED ON hat au.erdem keinen Einflu. auf INDEX oder REINDEX.
RECALL ALL ber.cksichtigt SET DELETED ON und setzt dann keine L.schmarkierungen zur.ck.

Hinweise

. Filtern von gel.schten Datens.tzen in einem einzelnen Arbeitsbereich:
Wenn Sie nur die gel.schten Datens.tze eines einzelnen Arbeitsbereiches filtern wollen, erreichen Sie dies mit SET FILTER TO .NOT. DELETED().

Beispiele

Das folgende Beispiel zeigt den Effekt von SET DELETED:

```
USE Verkauf NEW
? LASTREC()         // Ergebnis: 84
//
DELETE RECORD 4
COUNT TO nAnzahl
? nAnzahl           // Ergebnis: 84
SET DELETED ON
COUNT TO nAnzahl
? nAnzahl           // Ergebnis: 83
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET DELIMITERS Aktiviert, deaktiviert oder definiert die Begrenzer f.r GETs

---

Syntax

SET DELIMITERS TO [<cBegrenzer> | DEFAULT]
SET DELIMITERS on | OFF | <xlSchalter>

Argumente

ON aktiviert die Anzeige von Begrenzungszeichen f.r GET-Variablen.

OFF unterdr.ckt die Anzeige von Begrenzungszeichen f.r GET-Variablen.

<xlSchalter> ist ein logischer Ausdruck, der eingeklammert werden mu..
Dabei hat true (.T.) die gleiche Bedeutung wie ON und false (.F.)
entspricht OFF.

TO <cBegrenzer> definiert ein oder zwei Begrenzungszeichen. Wenn Sie nur ein Zeichen angeben, wird dieses Zeichen sowohl als Anfangs- als auch als Endbegrenzer benutzt. Bei der Angabe von zwei Zeichen wird das erste Zeichen als Anfangsbegrenzer und das zweite Zeichen als Endbegrenzer verwendet. Wenn Sie DEFAULT oder keine Begrenzungszeichen angeben, wird das Vorgabezeichen, der Doppelpunkt, als Begrenzer eingesetzt.

Anwendung

SET DELIMITERS erf.llt als Befehl zwei Zwecke. Zum einen definiert er die zu verwendenden Begrenzungszeichen und zum anderen schaltet er die Anzeige dieser Zeichen als Begrenzer von GET-Eingabefeldern ein bzw. aus. Der Befehl @...GET kann Begrenzer anzeigen, die vor und nach dem GET-Objekt plaziert werden. Wenn SET DELIMITERS auf ON gesetzt ist, wird die Anzeige des GET-Objektes um zwei Zeichen erweitert.

Die Begrenzungszeichen k.nnen mit der Klausel TO <cBegrenzer> bestimmt werden. Die Vorgabe ist der Doppelpunkt (:). Anfangs- und Endbegenzer k.nnen verschiedene Zeichen sein. Wenn Sie einen oder beide Begrenzer unterdr.cken m.chten, geben Sie ein Leerzeichen als entsprechendes Begrenzungszeichen an.

In der Regel sind Begrenzer nicht notwendig, da GET-Eingabefelder sowieso bereits farblich abgesetzt sind. Sie werden invers oder, bei Einstellung von SET INTENSITY auf ON, in der Farbe <enhanced> dargestellt.

Beispiele

Dieses Beispiel definiert als Begrenzungszeichen den Doppelpunkt sowie das Leerzeichen f.r das erste GET-Feld und die rechteckigen Klammern f.r das zweite GET-Eingabefeld:

```
LOCAL cVar := SPACE(5), cVar2 := SPACE(5)
SET DELIMITERS ON
SET DELIMITERS TO ": "
@ 1, 0 SAY "Eingabe" GET cVar
SET DELIMITERS TO "[]"
@ 2, 0 SAY "Eingabe" GET cVar2
READ
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET DEVICE Leitet die Ausgabe von @...SAY auf den Bildschirm oder den Drucker um

---

Syntax

SET DEVICE TO SCREEN | printer

Argumente

SCREEN leitet alle @...SAY-Ausgaben auf den Bildschirm und ist unabh.ngig von den Einstellungen f.r SET PRINTER und SET CONSOLE.

PRINTER leitet alle @...SAY-Ausgaben auf die mit SET PRINTER TO definierten Ausgabeeinheit. Dies k.nnen neben einem lokalen Drucker auch ein Netzwerk-Spooler oder eine Datei sein.

Anwendung

SET DEVICE leitet die Ausgaben von @...SAY entweder auf den Bildschirm oder auf die mit SET PRINTER TO definierten Ausgabeeinheit um. Ist SET DEVICE auf TO PRINTER eingestellt, erfolgt die Ausgabe nicht auf dem Bildschirm, sondern auf der mit SET PRINTER TO definierten Ausgabeeinheit. Die Einstellung von SET MARGIN wird von @...SAY-Befehlen ber.cksichtigt.

Bei der Ausgabe von @...SAY auf dem Drucker f.hrt Clipper immer dann einen EJECT (Seitenvorschub) durch, wenn die aktuelle Zeilennummer des Druckkopfes kleiner als die letzte Druckzeile ist. EJECT setzt PROW() und PCOL() auf Null zur.ck. Mit SETPRC() k.nnen Sie PROW() und PCOL() auf andere Werte setzen.
Die Ausgabe von @...SAY-Befehlen in eine Datei erreichen Sie durch die Kombination der Befehle SET PRINTER TO <xcDatei> und SET DEVICE TO PRINTER.

Beispiele

. Das folgende Beispiel leitet @...SAY zum Drucker um:

```
SET DEVICE TO PRINTER
@ 2, 10 SAY "Hallo allerseits"
EJECT
```

. Dieses Beispiel zeigt die Ausgabe von @...SAY in eine Datei:

```
SET PRINTER TO Output.TXT
SET DEVICE TO PRINTER
@ 10, 10 SAY "Datei ist: Output.TXT"
@ 11, 10 SAY DATE()
SET PRINTER TO             // Ausgabedatei schlie.en
SET DEVICE TO SCREEN
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET EPOCH Kontrolliert die Interpretation von Datumsangaben ohne Jahrhundertziffern

---

Syntax

SET EPOCH TO <nJahr>

Argumente

<nJahr> gibt das Basisjahr f.r eine 100-Jahre-Periode an, in die alle Datumsangaben, die nur zwei Jahresziffern enthalten, fallen sollen.

Anwendung

SET EPOCH ist ein Environment-Befehl, der die Interpretation von Datums-Zeichenketten mit nur zwei Jahresziffern festlegt. Die Umwandlung solcher Zeichenketten erfolgt, indem die Jahresziffern mit den Jahresziffern von <nJahr> verglichen werden. Wenn die Jahresziffern der Zeichenkette gr..er oder gleich den Jahresziffern in <nJahr> sind, wird angenommen, da. das Datum in das gleiche Jahrhundert wie <nJahr> f.llt. Anderenfalls wird das Datum dem folgenden Jahrhundert zugeordnet.

Der Vorgabewert f.r SET EPOCH ist 1900, so da. alle Datumsangaben ohne Jahrhundertziffern in das zwanzigste Jahrhundert eingeordnet werden.

Clipper unterst.tzt Daten vom 01.01.0100 bis zum 31.12.2999.

Beispiele

Das folgende Beispiel zeigt die Wirkung von SET EPOCH:

```
SET DATE FORMAT TO "dd.mm.yyyy"
? CTOD("27.05.1904")  // Ergebnis: 27.05.1904
? CTOD("27.05.67")    // Ergebnis: 27.05.1967
? CTOD("27.05.04")    // Ergebnis: 27.05.1904
//
SET EPOCH TO 1960
? CTOD("27.05.1904")  // Ergebnis: 27.05.1904
? CTOD("27.05.67")    // Ergebnis: 27.05.1967
? CTOD("27.05.04")    // Ergebnis: 27.05.2004
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

AFIELDS()* .bertr.gt eine Datenbankstruktur des aktuellen Arbeitsbereiches in Arrayvariablen

---

Syntax

```
AFIELDS([<aFeldnamen>],
        [<aFeldtypen>],
        [<aFeldl.ngen>],
        [<aDezimalstellen>]) ->nAnzahlFelder
```

Parameter

<aFeldnamen> ist das Array, das mit den Feldnamen gef.llt wird. Jedes gef.llte Element hat dann den Typ Zeichen.

<aFeldtypen> ist das Array, das mit den Feldtypen gef.llt wird. Jedes gef.llte Element hat dann den Typ Zeichen.

<aFeldl.ngen> ist das Array, das mit den Feldl.ngen gef.llt wird. Jedes gef.llte Element hat dann den Typ Numerisch.

<aDezimalstellen> ist das Array, das mit den Nachkommastellen der Felder gef.llt wird. Jedes gef.llte Element hat dann den Typ Numerisch. Ist der Feldtyp nicht numerisch, erh.lt das entsprechende Element den Wert 0.

R.ckgabe

AFIELDS() gibt die Anzahl der Felder oder die L.nge des k.rzesten Array-Parameters wieder. Werden keine Parameter angegeben oder ist im aktuellen Arbeitsbereich keine Datenbank ge.ffnet, ist der R.ckgabewert 0.

Anwendung

AFIELDS() f.llt eine Reihe von Arrays mit Strukturinformationen der im aktuellen Arbeitsbereich ge.ffneten Datei. F.r jedes Feld wird ein Arrayelement ben.tigt. AFIELDS() arbeitet .hnlich wie ADIR(), wobei ebenfalls eine Reihe von Arrays mit Informationen gef.llt werden. Bevor AFIELDS aufgerufen werden kann, m.ssen die Arrays, welche mit Strukturinformationen gef.llt werden sollen, angelegt werden; jedes Array mit der gleichen Anzahl von Elementen, wie Felder in der Datenbank vorhanden sind. Zur Ermittlung der Gr..e kann die Funktion FCOUNT() benutzt werden.

Wird nichts anderes angegeben, liefert AFIELDS() Strukturinformationen des aktuellen Arbeitsbereiches. Durch Angabe des Aliasnamens kann auch die Strukturinformation eines nicht selektierten Arbeitsbereiches ermittelt werden (siehe Beispiel).

AFIELD() ist nur aus Kompatibilit.tsgr.nden implementiert und wird f.r neue Applikationen daher nicht empfohlen. Die Funktion wird durch die Funktion DBSTRUCT() ersetzt, welche keine vordimensionierten Arrays ben.tigt und als Ergebnis ein mehrdimensionales Array mit Strukturinformationen liefert.

Beispiele

. Das folgende Beispiel zeigt den Gebrauch von AFIELDS() und ACHOICE(), um ein Men. zu erzeugen, aus dem Sie Felder ausw.hlen k.nnen:

```
USE verkauf IN 0
PRIVATE aFelder[FCOUNT()]
AFIELDS(aFelder)
@  1, 0 TO 10,11 DOUBLE
nAuswahl := ACHOICE(2, 1, 9, 10, aFelder)
@ 12, 0 SAY IF(nAuswahl <> 0, afelder[nAuswahl],;
   "kein Feld ausgew.hlt")
RETURN
```

. Nachfolgend ein Beispiel, das den Aliasnamen benutzt, um Arrays mit Strukturinformationen von VERKAUF.DBF zu f.llen:

```
LOCAL aFeldnamen, aFeldtypen, aLaenge, aDezimal
```

```
USE Verkauf NEW
USE Kunden NEW
*
aFeldnamen  := Verkauf->(ARRAY(FCOUNT()))
aFeldtypen  := Verkauf->(ARRAY(FCOUNT()))
aLaengen    := Verkauf->(ARRAY(FCOUNT()))
aDezimal    := Verkauf->(ARRAY(FCOUNT()))
*
Verkauf->(AFIELDS(aFeldnamen, aFeldtypen, aLaenge, aDezimal))
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

SET ESCAPE Schaltet [Esc] als Abschlu.taste f.r READ ein/aus

Syntax

SET ESCAPE ON | off | <xlSchalter>

Argumente

ON aktiviert [Esc] als Abschlu.taste f.r READ.

OFF sperrt [Esc] als Abschlu.taste f.r READ.

<xlSchalter> ist ein logischer Ausdruck, der eingeklammert werden mu..
Dabei hat true (.T.) die gleiche Bedeutung wie ON und false (.F.)
entspricht OFF.

Anwendung

Wenn Sie SET ESCAPE auf ON einstellen, k.nnen Sie den aktuellen READ-Proze. durch Dr.cken von [Esc] beenden. Alle .nderungen des aktuellen GET-Objektes, d.h. des augenblicklichen GET-Eingabefeldes, werden ignoriert, und die .berpr.fung durch RANGE oder VALID wird .bergangen. Falls SET ESCAPE auf OFF gesetzt ist, wird das Dr.cken von [Esc] ignoriert. Die Funktion der [Esc]-Taste kann aber .ber eine SET KEY-Prozedur umdefiniert werden, wobei die Einstellung von SET ESCAPE keine Rolle spielt.

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET EXACT* Schaltet die absolute .bereinstimmung beim Vergleich von Zeichenketten ein/aus

---

Syntax

SET EXACT on | OFF | <xlSchalter>

Argumente

ON aktiviert die absolute .bereinstimmung einschlie.lich der L.nge beim Vergleich von Zeichenketten.

OFF aktiviert den normalen Vergleich von Zeichenketten.

<xlSchalter> ist ein logischer Ausdruck, der eingeklammert werden mu.. Dabei hat true (.T.) die gleiche Bedeutung wie ON und false (.F.) entspricht OFF.

Anwendung

SET EXACT legt fest, auf welche Art zwei Zeichenketten .ber die relationalen Operatoren (=, >, <, =>, =<) miteinander verglichen werden. Ist SET EXACT auf OFF eingestellt, finden beim Vergleich folgende Regeln Anwendung, wobei angenommen wird, da. der Vergleich der zwei Zeichenketten cLinks und cRechts erfolgt:

. Wenn cRechts ein Nullstring ("") ist, wird true (.T.) zur.ckgegeben.

. Wenn LEN(cRechts) gr..er als LEN(cLinks) ist, wird false (.F.) zur.ckgegeben.

. In allen anderen F.llen werden alle Zeichen von cRechts mit denen in cLinks verglichen. Wenn alle Zeichen in cRechts gleich mit denen in cLinks sind, wird true (.T.) zur.ckgegeben, anderenfalls false (.F.).

Wenn SET EXACT auf ON eingestellt ist, geben alle relationalen Operatoren, abgesehen vom doppelten Gleichheitszeichen ==, nur dann true (.T.) zur.ck, wenn zwei Zeichenketten auch in der L.nge .bereinstimmen. Nachfolgende Leerstellen werden von diesen Operatoren allerdings nicht ber.cksichtigt. Diesen noch genaueren Vergleich f.hrt nur der Operator == durch.

SET EXACT ist ein Kompatibilit.ts-Befehl und wird daher nicht zum Einsatz in neuen Applikationen empfohlen.

Hinweise

Kompatibilit.t: Der Einsatz von SET EXACT hat in Clipper 5.0, anders als in anderen dBASE-Dialekten, nur einen Einflu. auf die relationalen Operatoren. Andere Operatoren werden von SET EXACT in Clipper 5.0 nicht betroffen. Das gilt auch f.r Suchbefehle, wie SEEK oder FIND. Um hier einen genauen, d.h. die L.nge des Zeichenschl.ssels mit einbeziehenden Vergleich durchzuf.hren, k.nnen Sie die als Beispiel beim SEEK-Befehl angef.hrte benutzer-definierte Funktion SucheGenau() verwenden.

Beispiele

Das folgende Beispiel zeigt die verschiedenen Ergebnisse von SET EXACT beim Vergleich mit dem Operator =.

```
SET EXACT OFF
? "123" = "12345" // Ergebnis: .F.
? "12345" = "123" // Ergebnis: .T.
? "123" = ""      // Ergebnis: .T.
? "" = "123"      // Ergebnis: .F.
? "123" = "123  " // Ergebnis: .F.
//
SET EXACT ON
? "123" = "12345" // Ergebnis: .F.
? "12345" = "123" // Ergebnis: .F.
? "123" = ""      // Ergebnis: .F.
? "" = "123"      // Ergebnis: .F.
```

```
? "123" = "123  " // Ergebnis: .T.
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET EXCLUSIVE* Aktiviert den shared- oder exclusive-Modus beim USE-Befehl f.r eine Datenbank

---

Syntax

SET EXCLUSIVE ON | off | <xlSchalter>

Argumente

ON bewirkt, da. Datenbanken im exclusive-Modus (= non-shared) ge.ffnet werden.

OFF bewirkt, da. Datenbanken im shared-Modus ge.ffnet werden.

<xlSchalter> ist ein logischer Ausdruck, der eingeklammert werden mu.. Dabei hat true (.T.) die gleiche Bedeutung wie ON und false (.F.) entspricht OFF.

Anwendung

SET EXCLUSIVE legt in einer Netzwerkumgebung fest, ob bei Verwendung des USE-Befehles ohne den Zusatz EXCLUSIVE oder SHARED die .ffnung von Datenbank-, Memo- und Indexdateien automatisch im exclusive-Modus erfolgt oder nicht. Wenn Sie eine Datei im exclusive-Modus .ffnen, kann sie im Netzwerk nicht gleichzeitig von anderen Benutzern mit USE ge.ffnet werden, bis Sie die Datei wieder schlie.en. Im exclusive-Modus sind daher Datei- und Datensatzsperren nicht notwendig.

Wenn SET EXCLUSIVE auf ON (= Voreinstellung) eingestellt ist, werden durch USE alle Datenbank- sowie die zugeh.rigen Dateien in einem non-shared-Modus (d.h. exclusive vom aktuellen Benutzer) ge.ffnet, es sei denn, der USE-Befehl wird mit dem Zusatz SHARED angegeben. Der Zusatz EXCLUSIVE wird nur f.r Dateioperationen ben.tigt, die den vollst.ndigen, uneingeschr.nkten Zugriff auf eine Datei voraussetzen, z.B. PACK, REINDEX oder ZAP.

Wenn SET EXCLUSIVE auf OFF eingestellt ist, werden durch USE alle Datenbanken sowie die zugeh.rigen Dateien im shared-Modus (d.h. andere Benutzer k.nnen die Datei ebenfalls bearbeiten und ver.ndern) ge.ffnet, es sei denn, der USE-Befehl wird mit dem Zusatz EXCLUSIVE angegeben. Der Zugriff von anderen Benutzern mu. im Programm mit RLOCK() und FLOCK() kontrolliert werden.

SET EXCLUSIVE ist ein Kompatibilit.ts-Befehl und wird daher nicht zum Einsatz in neuen Applikationen empfohlen. Er wird durch die Zus.tze EXCLUSIVE und SHARED beim USE-Befehl ersetzt.

Weitere Informationen zur Programmierung von Netzwerkanwendungen finden Sie im Handbuch Programmierung und Hilfsprogramme.

Hinweise

. Fehlerbehandlung: Der Versuch, eine bereits von einem anderen Benutzer mit EXCLUSIVE ge.ffnete Datenbank mit USE zu .ffnen, f.hrt zu einem Laufzeitfehler, wobei NETERR() auf true (.T.) gesetzt wird. Wenn die Kontrolle an die Stelle im Programm zur.ckverzweigt, an welcher der .ffnungsfehler aufgetreten ist, kann mit NETERR() das Ergebnis von USE abgefragt werden.

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET FILTER Verbirgt Datens.tze, welche die angegebene Filterbedingung nicht erf.llen

---

Syntax

SET FILTER TO <lBedingung>

Argumente

<lBedingung> ist ein logischer Ausdruck, der eine bestimmte Gruppe von Datens.tzen im aktuellen Arbeitsbereich zur Verarbeitung zul..t.

SET FILTER TO ohne Argument deaktiviert die augenblickliche Filterbedingung im aktuellen Arbeitsbereich.

Anwendung

Wenn Sie einen Filter setzen, verh.lt sich der aktuelle Arbeitsbereich so, als ob er nur die Datens.tze enth.lt, welche die Filterbedingung <lBedingung> erf.llen. Jeder Arbeitsbereich besitzt seine eigene Filterbedingung. Sie k.nnen eine existierende Filterbedingung mit der Funktion DBFILTER() abfragen, sie gibt die <lBedingung> als Zeichenkette zur.ck.

Die meisten Befehle und Funktionen, die den Datensatzzeiger bewegen, ber.cksichtigen den aktuellen Filter. Befehle, die einen Datensatz direkt .ber seine Satznummer ansprechen, lassen den Filter allerdings au.er acht. Dazu geh.ren GOTO, Befehle mit der RECORD-Klausel und mit SET RELATION verbundene Arbeitsbereiche, wenn .ber einen numerischen Ausdruck verbundene Arbeitsbereiche keinen kontrollierenden Index enthalten.

Wenn Sie einen Filter setzen, wird dieser erst aktiv, wenn der Datensatzzeiger von seiner aktuellen Position bewegt worden ist. Der Filter wird in der Regel mit GO TOP aktiviert.

Genauso wie SET DELETED hat auch SET FILTER keinen Einflu. auf INDEX und REINDEX.

Achtung

Obwohl SET FILTER den Eindruck erweckt, als ob im aktuellen Arbeitsbereich nur die durch den Filter sichtbare Menge an Datens.tzen vorhanden ist, werden dennoch immer alle Datens.tze sequentiell verarbeitet. Daher dauert die Verarbeitung einer gefilterten Datenbank immer genauso lange wie die einer ungefilterten Datenbank.

Beispiele

Das folgende Beispiel filtert in Personal.DBF die Datens.tze heraus, in denen Mitarbeiter stehen, die .lter als 50 Jahre sind:

```
USE Personal INDEX Name NEW
SET FILTER TO Alter > 50
LIST Nachname, Vorname, Alter, Telefon
SET FILTER TO
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET FIXED Schaltet die Anzeige einer festen Anzahl von Dezimalstellen bei Ausgaben ein/aus

---

Syntax

SET FIXED on | OFF | <xlSchalter>

Argumente

ON fixiert die Ausgabe von Dezimalstellen auf die mit SET DECIMALS festgelegte Anzahl.

OFF gibt der jeweiligen Operation oder Funktion die Kontrolle .ber die Anzahl der auszugebenden Dezimalstellen.

<xlSchalter> ist ein logischer Ausdruck, der eingeklammert werden mu.. Dabei hat true (.T.) die gleiche Bedeutung wie ON und false (.F.) entspricht OFF.

Anwendung

SET FIXED schaltet die Ausgabe von Dezimalstellen entsprechend der bei SET DECIMALS festgelegten Anzahl ein bzw. aus. Ist SET FIXED auf ON eingestellt, werden alle numerischen Werte mit den bei SET DECIMALS angegebenen Dezimalstellen angezeigt (der Vorgabewert von SET DECIMALS betr.gt zwei). Wenn SET FIXED auf OFF eingestellt ist, erfolgt die Ausgabe von numerischen Ausdr.cken entsprechend den im Kapitel Grundkonzepte dargelegten allgemeinen Regeln.

Sowohl der SET FIXED- als auch der SET DECIMALS-Befehl ber.hren nicht die Rechengenauigkeit, sondern nur das Ausgabeformat.

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET FORMAT* Aktiviert eine Formatprozedur bei der Ausf.hrung von READ

Syntax

SET FORMAT TO [<idProzedur>[.<DateiErweiterung>]]

Argumente

<idProzedur> ist der Name einer .FMT- oder .PRG-Datei oder einer Prozedur.

<DateiErweiterung> ist die Erweiterung der Formatdatei. Falls keine andere Erweiterung angegeben ist, wird die Standard-Erweiterung .FMT hinzugef.gt.

SET FORMAT TO ohne Argument deaktiviert das aktuelle Format.

Anwendung

SET FORMAT gibt eine Prozedur an, die vor einem READ auszuf.hren ist. Anders als in einer Interpreter-Umgebung werden Formate nicht w.hrend der Laufzeit einer Applikation ge.ffnet und ausgef.hrt. Stattdessen behandelt Clipper einen SET FORMAT-Befehl genau wie eine DO-Anweisung. Der Compiler pr.ft zuerst, ob er bereits eine Prozedur mit Namen <idProzedur> compiliert hat. Falls dies zutrifft, wird intern auf diese Prozedur verwiesen. Wird <idProzedur> nicht gefunden, pr.ft Clipper, ob auf der Platte eine Datei namens <idProzedur> existiert. Ist diese Datei nicht vorhanden, wird eine externe Referenz erzeugt, die zur Linkzeit aufgel.st werden mu..

SET FORMAT ist ein Kompatibilit.ts-Befehl und wird daher nicht zum Einsatz in neuen Applikationen empfohlen.

Hinweise

. Aktive Formatprozeduren: Clipper kann im Gegensatz zu anderen dBASE-Dialekten, die ein aktives Format f.r jeden Arbeitsbereich unterst.tzen, nur insgesamt ein Format (f.r alle Arbeitsbereiche) verwalten.

. L.schen des Bildschirms: Clipper l.scht den Bildschirm vor der Ausf.hrung einer Formatprozedur nicht.

. G.ltige Anweisungen: Zus.tzlich zu @...SAY und @...GET unterst.tzt Clipper alle g.ltigen Anweisungen und Befehle in Formatprozeduren.

. Mehrere Seiten: Clipper unterst.tzt keine Formatprozeduren, die sich .ber mehrere Seiten erstrecken.

Beispiele

Dieses Beispiel benutzt eine Formatprozedur, um Datens.tze an eine Datenbank anzuh.ngen, bis der Benutzer [Esc] dr.ckt:

```
USE Verkauf NEW
SET FORMAT TO VerkMask
DO WHILE LASTKEY() <> 27
  APPEND BLANK
  READ
ENDDO
RETURN
PROCEDURE VerkMask
  @ 12, 12 SAY "Filiale  : " GET  Filiale
  @ 13, 12 SAY "Verk.ufer: " GET  Verkaeufer
RETURN
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET FUNCTION Weist einer Funktionstaste eine Zeichenkette zu

Syntax

SET FUNCTION <nFunktionsTaste> TO <cString>

Argumente

<nFunktionsTaste> gibt die Nummer der zu belegenden Funktionstaste an.

<cString> ist die Zeichenkette, die der Funktionstaste zugewiesen werden soll.

Anwendung

SET FUNCTION ordnet eine Zeichenkette einer Funktionstaste zu, deren Nummer von 1 bis 40 reichen kann. Bei Bet.tigung einer so umbelegten Funktionstaste wird die zugeh.rige Zeichenkette in den Tastatur-Eingabepuffer geschrieben. <cString> darf Steuerzeichen wie [Strg]+ [C] oder [Strg]+ [S] enthalten, um so z.B. Steuerungs- und Bearbeitungsvorg.nge aus einem Wartezustand heraus vorzunehmen.

Tabelle 4-15: Funktionstasten-.bersicht

Nummer der Funktionstaste	Tats.chliche Taste
1 - 10	F1 - F10
11 - 20	Umschalt+F1 - Umschalt+F10
21 - 30	Strg+F1 - Strg+F10
31 - 40	Alt+F1 - Alt+F10

Achtung

Clipper 5.0 wandelt SET FUNCTION durch den Pre-Prozessor in SET KEY- und KEYBOARD-Befehle um. Daher l.scht jede Zuweisung .ber SET FUNCTION eine ggfs. vorhandene SET KEY-Zuweisung f.r dieselbe Taste und umgekehrt. Dieses Verhalten ist inkompatibel zu den Vorg.ngerversionen, die separate Listen f.r Zuweisungen mit SET FUNCTION und SET KEY verwalten.

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET INDEX .ffnet eine oder mehrere Indexdateien im selektierten Arbeitsbereich

---

Syntax

SET INDEX TO [<xcIndexListe>]

Argumente

<xcIndexListe> ist eine Liste mit bis zu 15, durch Kommata voneinander getrennten Namen von Indexdateien (.NTX oder .NDX). Jeder Name, der einen Nullstring ("") ergibt oder leer ist, wird ignoriert. Ein Dateiname kann als Literal oder als eingeklammerter Zeichenausdruck angef.hrt werden. Falls keine andere Erweiterung angegeben ist, wird die Standard-Erweiterung hinzugef.gt, die vom verwendeten Datenbank-Treiber abh.ngt. Die Erweiterung ist .NTX f.r den Clipper-Datenbank-Treiber und .NDX f.r den Datenbank-Treiber f.r dBASE III PLUS.

SET INDEX TO ohne Argument schlie.t alle ge.ffneten Indexdateien im aktuellen Arbeitsbereich.

Anwendung

SET INDEX schlie.t im aktuellen Arbeitsbereich alle derzeit ge.ffneten Indexdateien und .ffnet die in der <xcIndexListe> aufgef.hrten Indexdateien. Bei mehreren ge.ffneten Indexdateien wird die zuerst angegebene Datei zum kontrollierenden, d.h. die Satzreihenfolge bestimmenden Index. Der Datensatzzeiger wird anf.nglich auf den ersten logischen Datensatz im Index positioniert. W.hrend der Bearbeitung einer Datenbank werden alle derzeit ge.ffneten Indizes automatisch aktualisiert, wenn ein Schl.sselwert ge.ndert oder angehangen wird. Der kontrollierende Index kann .ber den Befehl SET ORDER TO ver.ndert werden, ohne die jeweiligen Indexdateien mit SET INDEX in einer anderen Reihenfolge .ffnen zu m.ssen.

SET INDEX wird haupts.chlich benutzt, um in einer Netzwerkumgebung Indexdateien zu .ffnen, ohne den USE-Befehl mit der INDEX-Klausel zu verwenden. Normalerweise .ffnen Sie dabei die <xcDatenbank> mit USE und .berpr.fen, ob der Zugriff erfolgreich war. Trifft dies zu, werden die zugeh.rigen Indexdateien mit SET INDEX ge.ffnet (vgl. folgendes Beispiel).

Beispiele

. Das folgende Beispiel demonstriert den Einsatz von erweiterten Ausdr.cken zur .ffnung von Indexdateien. Jeder Dateiname wird dabei .ber eine separate Variable angegeben.

```
xcIndex1 := "Name"
xcIndex2 := "Nummer"
SET INDEX TO (xcIndex1), (xcIndex2)
```

. Dieses Beispiel zeigt die Verwendung von SET INDEX in einer Netzwerkumgebung:

```
USE Verkauf SHARED
IF .NOT. NETERR()
  SET INDEX TO Verkauf, Gebiet
ELSE
  ? "Datenbank konnte nicht ge.ffnet werden!"
  BREAK
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET INTENSITY Schaltet die enhanced-Anzeige von GETs und PROMPTs ein/aus

---

Syntax

SET INTENSITY ON | off | <xlSchalter>

Argumente

ON aktiviert sowohl die standard- als auch die enhanced-Farben f.r Ausgaben.

OFF deaktiviert enhanced-Farben f.r Ausgaben. F.r alle Anzeigen wird nur die standard-Farbe verwendet.

<xlSchalter> ist ein logischer Ausdruck, der eingeklammert werden mu.. Dabei hat true (.T.) die gleiche Bedeutung wie ON und false (.F.) entspricht OFF.

Anwendung

SET INTENSITY schaltet die enhanced-Anzeige von GETs und PROMPTs ein bzw. aus. Wenn SET INTENSITY auf OFF gestellt ist, erscheinen GETs und SAYs nur in der Farbe standard. Ist SET INTENSITY auf ON eingestellt, werden GET-Eingabefelder in der Farbe enhanced ausgegeben.

Wenn SET INTENSITY auf OFF gestellt ist, erscheinen alle Men.ausgaben mit PROMPT in der standard-Farbe, und der jeweils aktuelle PROMPT wird mit dem Cursor markiert. Wenn SET INTENSITY, wie vorgegeben, auf ON gestellt ist, wird der jeweils aktuelle PROMPT in der enhanced-Farbe markiert. Der Cursor ist dann nicht sichtbar.

SET INTENSITY hat keinen Einflu. auf ACHOICE() und DBEDIT().

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET KEY Ordnet einer Taste die Aktivierung einer Prozedur zu

Syntax

SET KEY <nInkeyTastencode> TO [<idProzedur>]

Argumente

<nInkeyTastencode> gibt den INKEY()-Code der zu belegenden Taste an.

<idProzedur> gibt den Namen der Prozedur an, die immer dann ausgef.hrt werden soll, wenn die zugewiesene Taste gedr.ckt worden ist. Falls <idProzedur> nicht mit angegeben wird, erh.lt die angef.hrte Taste ihre urspr.ngliche Funktion zur.ck.

Anwendung

SET KEY ist ein Tastaturbefehl, der die Ausf.hrung einer Prozedur aus einem Wartezustand heraus erlaubt, wenn die zugewiesene Taste gedr.ckt worden ist. Ein Wartezustand ist jeder Modus, der Eingaben von der Tastatur entgegennimmt. Dazu geh.ren z.B. ACHOICE(), DBEDIT(), MEMOEDIT(), ACCEPT, INPUT, READ und WAIT, aber nicht INKEY(). Nachdem eine umdefinierte Taste gedr.ckt worden ist, wird die angegebene Prozedur ausgef.hrt, wobei ihr drei Parameter .bergeben werden. Diese Parameter entsprechen PROCNAME(), PROCLINE() und READVAR(). Die Namen der Prozedur und Variablen sind vom Typ Zeichen, die Zeilennummer ist numerisch.

Es k.nnen maximal 32 Tasten gleichzeitig umbelegt werden. Bei Programmstart ist die Prozedur "Help" automatisch der Taste [F1] zugewiesen. Wenn Sie eine Prozedur gleichen Namens in eine Anwendung einbinden, ruft [F1] Ihre Prozedur auf.

SET KEY-Prozeduren sollten so aufgebaut sein, da. sie nach ihrem Aufruf den aktuellen Programmzustand (Bildschirminhalt, aktueller Arbeitsbereich etc.) sichern und vor Beendigung wiederherstellen.

Achtung

Clipper 5.0 wandelt SET FUNCTION durch den Pre-Prozessor in SET KEY- und KEYBOARD-Befehle um. Daher l.scht jede Zuweisung .ber SET FUNCTION eine ggfs. vorhandene SET KEY-Zuweisung f.r dieselbe Taste und umgekehrt. Dieses Verhalten ist inkompatibel zu den Vorg.ngerversionen, die separate Listen f.r Zuweisungen mit SET FUNCTION und SET KEY verwalten.

Hinweise

. Vorrang: Definitionen mit SET KEY haben Vorrang vor SET ESCAPE und SETCANCEL().

. CLEAR in einer SET KEY-Prozedur: Der Bildschirm sollte aus einer SET KEY-Prozedur heraus nicht mit CLEAR gel.scht werden, da dann gleichzeitig alle GETs gel.scht werden und somit ein READ beendet wird. Benutzen Sie stattdessen CLEAR SCREEN oder CLS.

. Abschlu. eines READ aus einer SET KEY-Prozedur heraus: Die nachfolgende Tabelle zeigt verschiedene Wege, um ein READ aus einer SET KEY-Prozedur heraus zu beenden:

Tabelle 4-16: Abschlu. eines READ aus einer SET-KEY Proz.

Befehl	Aktion
CLEAR GETS	Beendet READ ohne Speicherung des aktuellen GET-Feldes
BREAK	Beendet READ ohne Speicherung des aktuellen GET-Feldes
KEYBOARD Strg+W	Beendet READ mit Speicherung des aktuellen GET-Feldes
KEYBOARD Esc	Beendet READ ohne Speicherung des aktuellen GET-Feldes

Beispiele

Das folgende Beispiel demonstriert den Einsatz von SET KEY, um w.hrend der Neuanlage eines Kunden .ber [F2] eine Auswahlliste mit Kundengruppen zu pr.sentieren:

```
#include "Inkey.CH"
```

```
//
SET KEY K_F2 TO KundenGruppen
USE KundGrup NEW
USE Kunden NEW
@ 10, 10 GET Kunden->KundenNr
//
<weitere GET-Anweisungen f.r die anderen Felder>...
//
READ
RETURN
PROCEDURE KundenGruppen(cProz, nZeile, CVar)
  IF cVar = "KundenNr"
    SAVE SCREEN
    KundGrup->(DBEDIT(10, 10, 18, 40, {"Kundengruppen"}))
    KEYBOARD CHR(K_CTRL_Y) + KundGrup->Typ + CHR(K_HOME)
    RESTORE SCREEN
  ELSE
    TONE(100,2)
  ENDIF
RETURN
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist Inkey.CH

See Also:

SET MARGIN Stellt den linken Rand f.r den Ausdruck ein

Syntax

SET MARGIN TO [<nLinkerRand>]

Argumente

<nLinkerRand> ist eine positive Zahl, welche die Anzahl von Spalten angibt, um die alle nachfolgenden Druckausgaben vom linken Seitenrand an einger.ckt werden sollen. Negative Werte setzen MARGIN auf Null zur.ck.

SET MARGIN TO ohne Argument setzt <nLinkerRand> auf Null zur.ck. Null ist auch die Vorgabe bei Programmstart.

Anwendung

SET MARGIN beeinflu.t die Druckausgabe sowohl von Konsolen-Befehlen als auch von @...SAY. Bei Konsolen-Befehlen wird jede auszudruckende Zeile um die Zeichenanzahl von <nLinkerRand> einger.ckt. Bei @...SAY-Befehlen wird <nLinkerRand> zum angegebenen Spaltenwert hinzuaddiert. SET MARGIN beeinflu.t die Bildschirmausgabe nicht.

Hinweis

Bei Druckbefehlen mit @...SAY und PCOL() mit gesetztem linken Rand wird <nLinkerRand> zu jeder Spaltennummer hinzuaddiert. Das geschieht deshalb, weil PCOL() immer die letzte Druckspalte unter Einbeziehung von <nLinkerRand> korrekt angibt. Daher sollte eine Einstellung von SET MARGIN zum Ausdruck von relativen Spaltenpositionen mit PCOL() vermieden werden.

Beispiele

Das folgende Beispiel stellt einen linken Druckrand von 5 Zeichen ein und druckt eine Liste der Datei Verkauf.DBF:

```
USE Verkauf NEW
SET MARGIN TO 5
LIST Filiale, Verkaeufer TO PRINTER
SET MARGIN TO
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist Inkey.CH

See Also:

SET MESSAGE Stellt die Ausgabezeile f.r @...PROMPT-Meldungen ein

---

Syntax

SET MESSAGE TO [<nZeile> [CENTER | CENTRE]]

Argumente

<nZeile> gibt die Ausgabezeile f.r die Meldungen an.

CENTER | CENTRE zentriert die Meldungen in der Ausgabezeile.

Die Angabe von SET MESSAGE TO 0 oder SET MESSAGE TO ohne Argument unterdr.ckt die Ausgabe von Meldungen.

Anwendung

SET MESSAGE ist ein Men.befehl, der die Ausgabezeile f.r die Meldungen des Befehles @...PROMPT definiert. Nach dem Start eines Clipper-Programmes ist als Ausgabezeile 0 vorgegeben, d.h es erfolgen keine Meldungsanzeigen. Meldungen werden ansonsten an der Position <nZeile> 0 ausgegeben, es sei denn, sie werden mit dem Zusatz CENTER zentriert.

Beispiele

Das folgende Beispiel zeigt ein einfaches Light-bar-Men. mit aktivierten und zentriert ausgegebenen Meldungen:

```
SET MESSAGE TO 23 CENTER
SET WRAP ON
@ 5, 5 PROMPT "Eins" MESSAGE "Auswahl 1"
@ 6, 5 PROMPT "Zwei" MESSAGE "Auswahl 2"
MENU TO nWahl
//
IF nWahl == 0
  EXIT
ELSEIF nWahl == 1
  Proz1()
ELSEIF nWahl == 2
  Proz2()
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET ORDER Weist einem offenen Index die Funktion des Kontrollindexes zu

---

Syntax

SET ORDER TO [<nIndex>]

Argumente

<nIndex> gibt den kontrollierenden Index an. <nIndex> zeigt auf die absolute Position innerhalb der ge.ffneten Indexdateien <xcIndexListe> und kann eine Zahl von 0 bis 15 sein.

Wenn <nIndex> nicht angegeben wird, wird SET ORDER auf Null eingestellt.

Anwendung

Wenn Sie mit SET ORDER einen neuen kontrollierenden Index angeben, werden weiterhin alle ge.ffneten Indizes korrekt aktualisiert, falls Sie Datens.tze .ndern oder anh.ngen. Das gilt auch f.r SET ORDER TO 0. Nach dem Wechsel des kontrollierenden Indexes bleibt der Datensatzzeiger auf seiner bisherigen Position.

SET ORDER TO 0 oder SET ORDER ohne Argument stellen die nat.rliche Reihenfolge der Datenbank (nach Satznummern) wieder her, ohne allerdings die derzeit ge.ffneten
Indexdateien zu schlie.en.

Die Funktion INDEXORD() gibt die Position des derzeit kontrollierenden Indexes als ganzzahligen numerischen Wert zur.ck.

Beispiele

Das folgende Beispiel zeigt einige Verwendungsm.glichkeiten des SET ORDER-Befehles:

```
USE Kunden NEW
INDEX ON Nachname TO Name
INDEX ON Ort + Bundesland TO Region
SET INDEX TO Name, Region
//
SET ORDER TO 2
? INDEXKEY(INDEXORD())  // Ergebnis: 2
//
SET ORDER TO 0
? INDEXKEY(INDEXORD())  // Ergebnis: 0
//
SET ORDER TO 1
? INDEXKEY(INDEXORD())  // Ergebnis: 1
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET PATH Definiert den Clipper-Suchpfad f.r den Zugriff auf Dateien

---

Syntax

SET PATH TO [<xcPfadSpezifikationsListe>]

Argumente

<xcPfadSpezifikationsListe> gibt eine Liste der Pfade an, in denen Clipper nach einer Datei suchen soll, wenn sie im aktuellen Verzeichnis nicht gefunden worden ist. Die Liste kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Die einzelnen Pfade k.nnen durch Kommata oder Semikolons voneinander getrennt werden. Die Fortf.hrung des SET PATH-Befehles mit einem Semikolon auf einer neuen Befehlszeile ist nicht m.glich, au.er Sie geben <xcPfadSpezifikation> als erweiterten Ausdruck an.

Anwendung

SET PATH gibt Befehlen und Funktionen, die Datenbanken und zugeh.rige Dateien .ffnen, die M.glichkeit, auch Dateien in anderen Pfaden als dem aktuellen Verzeichnis zu suchen und zu .ffnen. Dies erfolgt, indem die angegebene <xcPfadSpezifikationsListe> nach einer Datei durchsucht wird, die nicht im aktuellen oder im mit SET DEFAULT eingestellten Verzeichnis gefunden worden ist. Memo- und Low-Level-Funktionen ber.cksichtigen allerdings weder die SET DEFAULT- noch die SET PATH-Einstellungen.

Ein Pfad ist ein Zeiger auf ein Verzeichnis, der aus einem optionalen Laufwerksbezeichner mit nachfolgendem Doppelpunkt sowie einer Liste der vom Haupt- bis zum gew.nschten Zielverzeichnis reichenden, durch das Backslash-Zeichen \ voneinander getrennten Verzeichnissen besteht. Eine Pfadliste wiederum besteht aus mehreren hintereinander aufgef.hrten, durch Kommata oder Semikolons getrennten Pfaden.

Beim Zugriff auf eine Datei sucht Clipper zuerst im vorgegebenen Laufwerk und Verzeichnis, das nach dem Start eines Clipper-Programmes das aktuelle DOS-Laufwerk bzw. -Verzeichnis oder ein mit SET DEFAULT eingestellter Pfad ist. Wurde die Datei nicht gefunden, durchsucht Clipper der Reihe nach jeden Pfad in der <xcPfadSpezifikationsListe>, bis eine passende Datei gefunden worden ist.

Zur Erzeugung von neuen Dateien in einem anderen Verzeichnis oder Laufwerk verwenden Sie entweder SET DEFAULT TO <xcPfadSpezifikation> oder Sie geben den Pfadnamen explizit mit dem Namen der neuen Datei an.

SET PATH TO ohne Argument l.scht die aktuelle Pfadliste, so da. Clipper nur im vorgegebenen Verzeichnis sucht.

Beispiele

. Dieses Beispiel zeigt einen typischen SET PATH-Befehl:

```
SET PATH TO A:\Inventar;B:\Personal
```

. Dieses Beispiel zeigt, wie der Suchpfad w.hrend der Programmausf.hrung durch .bergabe einer DOS-Umgebungsvariablen, deren Inhalt mit GETENV() ermittelt wird, konfiguriert werden kann. Diesen so ermittelten Wert verarbeitet SET PATH.

```
SET CLIP_PATH=C:\APPS\DATEN,C:\APPS\PROGS
```

. Im Konfigurationsteil Ihrer Applikation m..te dann folgende Zeile stehen:

```
SET PATH TO (GETENV("CLIP_PATH"))
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET PRINTER Leitet die Ausgabe von Konsolen-Befehlen zus.tzlich auf den Drucker um oder definiert das Ziel von Ausdrucken

---

Syntax

SET PRINTER on | OFF | <xlSchalter>
SET PRINTER TO [<xcGer.teEinheit> | <xcDatei>]

Argumente

ON aktiviert die Ausgabe von Konsolen-Befehlen zum Drucker.

OFF unterdr.ckt die Ausgabe von Konsolen-Befehlen zum Drucker.

<xlSchalter> ist ein logischer Ausdruck, der eingeklammert werden mu.. Dabei hat true (.T.) die gleiche Bedeutung wie ON und false (.F.) entspricht OFF.

TO <xcGer.teEinheit> gibt den Namen derjenigen Ausgabeeinheit an, an die alle nachfolgenden Druckausgaben gesendet werden. Der Ger.tename kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Zus.tzlich kann die Ausgabeeinheit lokal oder im Netzwerk vorhanden sein. Wenn Sie eine nicht existierende <xcGer.teEinheit> angeben, wird eine Ausgabedatei gleichen Namens erzeugt. Sie m.ssen den Ger.tenamen ohne nachfolgenden Doppelpunkt angeben.

TO <xcDatei> leitet die Ausgabe in die Datei <xcDatei> um. <xcDatei> kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Falls keine andere Erweiterung angegeben ist, wird die Standard-Erweiterung .PRN hinzugef.gt.

SET PRINTER ohne Argument schlie.t die vorher gew.hlte Ausgabeeinheit oder -datei und w.hlt die Vorgabeeinheit.

Anwendung

SET PRINTER hat zwei verschiedene Argumentformen. Die Form on | OFF bestimmt, ob die Ausgabe von Konsolen-Befehlen zum Drucker gesendet werden soll oder nicht. Unter Konsolen-Befehlen werden dabei in der Regel Befehle verstanden, die Bildschirmausgaben ohne Angabe einer Zeilen- oder Spaltennummer vornehmen. Alle diese Befehle, bis auf ? | ??, k.nnen mit dem Zusatz TO PRINTER angegeben werden, so da. die Ausgaben zus.tzlich zum Drucker gesendet werden. Konsolen-Befehle werden immer auf dem Bildschirm ausgegeben, soweit nicht SET CONSOLE auf OFF eingestellt ist. SET PRINTER ON ber.hrt nicht die Ausgaben von @...SAY. Die Ausgaben dieses Befehles k.nnen durch SET DEVICE TO PRINTER zum Drucker gesendet werden.

In seiner zweiten Argumentform definiert SET PRINTER .ber den Zusatz TO das Ziel aller zum Drucker gesendeten Ausgaben. Dies betrifft diesmal auch die Ausgaben durch @...SAY, wenn SET DEVICE TO PRINTER eingestellt worden ist. Die Ausgabe kann entweder an eine Ger.teeinheit (LPT1, LPT2, LPT3, COM1, COM2 oder PRN) oder in eine Datei erfolgen.

Eine als Ziel angegebene Datei wird im vorgebenen Verzeichnis angelegt. Falls bereits eine Datei mit dem Namen der <xcDatei> existiert, wird diese Datei ohne Vorwarnung .berschrieben. Alle nachfolgenden Ausgaben zum Drucker werden stattdessen in diese Datei geschrieben, bis die Datei mit SET PRINTER TO ohne Argument geschlossen wird.

SET PRINTER TO kann f.r viele Zwecke eingesetzt werden, u.a.:

. Verwaltung von mehreren Druckern durch Wechsel der Druckerschnittstelle

. Ausdruck in eine Datei zur sp.teren Ausgabe auf einem Drucker oder zur .bermittlung via DF.

. Entleerung des Drucker-Spoolers und Zur.cksetzen der vorgegebenen Ausgabeeinheit

Hinweise

. Kompatibilit.t: Clipper unterst.tzt nicht die Syntax SET PRINTER TO \\SPOOLER oder \\CAPTURE. Wenn Sie diese Befehle angeben, werden die Ausgabedateien Spooler.PRN bzw. Capture.PRN erzeugt. Die Symbole \\ werden ignoriert.

. Dateiende-Marke: Bei Ausgabe in eine Datei wird kein Zeichen f.r das Dateiende (CHR(26)) angehangen, wenn die Datei geschlossen wird. Dies erreichen Sie durch Ausf.hrung des Befehles ?? CHR(26) vor dem SET PRINTER TO-Befehl, der die Ausgabedatei schlie.t.

. Netzwerke: Bei einigen Netzwerksystemen mu. der Drucker einer Arbeitsstation erst zum File-Server umgeleitet werden (normalerweise durch Ausf.hrung des Netzwerk-Spooler-Programmes).

Beispiele

. Dieses Beispiel sendet die Ausgaben von ? zum Drucker, wobei die Bildschirmausgabe durch SET CONSOLE OFF unterdr.ckt wird:

```
USE Kunden NEW
SET PRINTER ON
SET CONSOLE OFF
DO WHILE .NOT. EOF()
  ? Kunden->Name, Kunden->Telefon
  SKIP
ENDDO
EJECT
SET PRINTER OFF
SET CONSOLE ON
CLOSE
RETURN
```

. Das folgende Beispiel sendet die Ausgabe an LPT1 und leert den Drucker-Spooler nach Beendigung der Ausgabe:

```
SET PRINTER TO LPT1
//
<Druckanweisungen>...
//
SET PRINTER TO    // Leert den Drucker-Spooler
```

. Dieses Beispiel druckt in eine Textdatei, wobei eine existierende Datei .berschrieben wird:

```
SET PRINTER TO PrnDatei.TXT
SET DEVICE TO PRINTER
SET PRINTER ON
//
@ 0, 0 SAY "Das geht in PrnDatei.TXT"
? "Das auch!"
//
SET DEVICE TO SCREEN
SET PRINTER OFF
SET PRINTER TO    // Schlie.t die Ausgabedatei
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET PROCEDURE* Compiliert Prozeduren und Funktionen in die aktuelle .OBJ-Datei

---

Syntax

SET PROCEDURE TO [<xcProgrammDatei>[.<DateiErweiterung>]

Argumente

<xcProgrammDatei> gibt den Namen der Prozedurdatei an, die in die aktuelle Objektdatei compiliert werden soll. Der Name kann optional einen Pfad- oder Laufwerksbezeichner enthalten.

<DateiErweiterung> ist die optionale Erweiterung der Prozedurdatei. Wenn keine <DateiErweiterung> angegeben ist, wird .PRG angenommen.

SET PROCEDURE TO ohne Argument wird ignoriert.

Anwendung

SET PROCEDURE weist den Compiler an, alle in der angegebenen Prozedurdatei deklarierten Prozeduren und benutzer-definierten Funktionen in die aktuelle Objektdatei zu compilieren.

SET PROCEDURE ist ein Kompatibilit.ts-Befehl und wird daher nicht zum Einsatz in neuen Applikationen empfohlen. Er wird durch andere M.glichkeiten ersetzt, die der jeweiligen, compilierten Umgebung besser entsprechen. Dazu geh.ren u.a. die Compiler-Anweisung #include und die Skript-Datei mit der Erweiterung .CLP.

Weitere Informationen zu Programmaufbau und -konfiguration finden Sie im Kapitel
Compilieren und Linken des Handbuches Programmierung und Hilfsprogramme.

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET RELATION Verbindet zwei Arbeitsbereiche .ber einen Schl.sselausdruck oder die Satznummern miteinander

---

Syntax

SET RELATION TO [<Schl.sselAusdruck> | <nDatensatz> INTO <xcAlias>]
[,<Schl.sselAusdruck2> | <nDatensatz2> INTO <xcAlias2>...] [ADDITIVE]

Argumente

TO <Schl.sselAusdruck> ist ein Ausdruck, der dazu verwendet wird, eine Suche im verkn.pften Arbeitsbereich durchzuf.hren. Die Suche erfolgt immer dann, wenn der Datensatzzeiger im f.hrenden Arbeitsbereich bewegt wird. Zur korrekten Durchf.hrung mu. im verkn.pften Arbeitsbereich ein kontrollierender Index ge.ffnet sein.

TO <nDatensatz> ist ein Ausdruck, der dazu verwendet wird, den Datensatzzeiger im verkn.pften Arbeitsbereich auf die entsprechende Satznummer zu bewegen (entspricht GOTO). Die Positionierung erfolgt immer dann, wenn der Datensatzzeiger im f.hrenden Arbeitsbereich bewegt wird. Wenn f.r <nDatensatz> RECNO() angegeben wird, verschiebt sich der Datensatzzeiger im verkn.pften Arbeitsbereich immer auf die gleiche Satznummer wie im f.hrenden Arbeitsbereich. Damit die Relation .ber einen numerischen Ausdruckstyp korrekt funktioniert, darf im verkn.pften Arbeitsbereich kein kontrollierender Index in Benutzung sein.

INTO <xcAlias> gibt den verkn.pften Arbeitsbereich an. <xcAlias> kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden.

ADDITIVE f.gt die angegebene Relation zu den im aktuellen Arbeitsbereich bereits bestehenden Relationen hinzu. Wenn dieser Zusatz nicht angegeben wird, werden alle existierenden Relationen gel.st und die im aktuellen Befehl definierte Relation aktiviert.

SET RELATION ohne Argument l.st alle bestehenden Relationen im aktuellen Arbeitsbereich.

Anwendung

SET RELATION ist ein Datenbankbefehl, der einen f.hrenden Arbeitsbereich .ber einen <Schl.sselAusdruck>, numerischen Ausdruck oder die Satznummer mit einem oder mehreren verkn.pften Arbeitsbereichen verbindet. Ein f.hrender Arbeitsbereich kann mit maximal acht Arbeitsbereichen verbunden sein. Eine Relation bedeutet, da. der Datensatzzeiger im verkn.pften Arbeitsbereich immer dann entsprechend der angegebenen Verkn.pfungsbedingung bewegt wird, wenn der Datensatzzeiger im f.hrenden Arbeitsbereich bewegt worden ist. Wird im verkn.pften Arbeitsbereich kein passender Satz gefunden, wird der Datensatzzeiger auf LASTREC()+1 positioniert, EOF() gibt true (.T.) und FOUND() false (.F.) zur.ck.

Die Verbindungsmethode von f.hrendem und verkn.pftem Arbeitsbereich h.ngt vom Typ des <Schl.sselAusdruckes> und der Existenz eines kontrollierenden Indexes im verkn.pften Arbeitsbereich ab. Ist im verkn.pften Arbeitsbereich ein kontrollierender Index aktiv, erfolgt die Suche .ber einen normalen SEEK-Vorgang. Gibt es im verkn.pften Arbeitsbereich keinen kontrollierenden Index und ist der <Schl.sselAusdruck> ein numerischer Typ, wird im verkn.pften Arbeitsbereich stattdessen ein GOTO ausgef.hrt.

Hinweise

. Zyklische Relationen: Ein f.hrender Arbeitsbereich kann weder direkt noch indirekt mit sich selbst verbunden sein.

. SOFTSEEK: SET RELATION verh.lt sich immer so, als ob SET SOFTSEEK OFF ist (selbst dann, wenn SET SOFTSEEK explizit auf ON eingestellt worden ist). Daher wird der Datensatzzeiger im verkn.pften Arbeitsbereich immer auf LASTREC()+1 gesetzt, wenn kein passender Satz gefunden worden ist.

. Relationen .ber Satznummern: Die Verbindung zweier Arbeitsbereiche .ber die jeweils gleiche Satznummer erfolgt, indem Sie RECNO() f.r

<nDatensatz> angeben. Au.erdem darf im verkn.pften Arbeitsbereich kein kontrollierender Index aktiv sein.

Beispiele

. Dieses Beispiel verbindet drei Arbeitsbereiche in einer mehrfach verkn.pften Relation, wobei Kunden sowohl mit Rechnung als auch PLZ verbunden ist:

```
USE Rechnung INDEX Rechnung NEW
USE PLZ INDEX PLZ NEW
USE Kunden NEW
SET RELATION TO KundenNr INTO Rechnung, PLZ INTO PLZ
LIST Kunde, PLZ->Ort, Rechnung->Nummer, Rechnung->Umsatz
```

. Sp.ter k.nnen Sie im Programm mit dem Zusatz ADDITIVE eine weitere Relation hinzuf.gen:

```
USE Storno INDEX Storno NEW
SELECT Kunden
SET RELATION TO KundenNr INTO Storno ADDITIVE
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET SCOREBOARD Schaltet die Meldungsausgabe durch READ und MEMOEDIT() ein/aus

---

Syntax

SET SCOREBOARD ON | off | <xlSchalter>

Argumente

ON erlaubt die Ausgabe von Meldungen durch READ und MEMOEDIT() in der Bildschirmzeile Null.

OFF unterdr.ckt diese Meldungen.

<xlSchalter> ist ein logischer Ausdruck, der eingeklammert werden mu.. Dabei hat true (.T.) die gleiche Bedeutung wie ON und false (.F.) entspricht OFF.

Anwendung

SET SCOREBOARD kontrolliert, ob Meldungen von READ und MEMOEDIT() in der Bildschirmzeile Null angezeigt werden oder nicht. Wenn SET SCOREBOARD auf ON eingestellt ist, gibt READ Meldungen bei RANGE-Fehlern, ung.ltigen Datumswerten und f.r den Einf.gemodus aus. MEMOEDIT() zeigt eine Abfragemeldung f.r den Abbruch sowie den Einf.gemodus an.

Die automatische Ausgabe dieser Meldungen wird mit SET SCOREBOARD OFF unterdr.ckt.

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET SOFTSEEK Schaltet die relative Suche mit SEEK ein/aus

Syntax

SET SOFTSEEK on | OFF | <xlSchalter>

Argumente

ON bewirkt, da. der Datensatzzeiger auf den n.chsten Datensatz mit einem h.heren Schl.sselwert positioniert wird, nachdem ein Suchproze. erfolglos war.

OFF bewirkt, da. der Datensatzzeiger auf EOF() positioniert wird, nachdem ein Suchproze. erfolglos war.

<xlSchalter> ist ein logischer Ausdruck, der eingeklammert werden mu.. Dabei hat true (.T.) die gleiche Bedeutung wie ON und false (.F.) entspricht OFF.

Anwendung

SET SOFTSEEK aktiviert eine relative Suche. Das bedeutet, da. auch dann ein Datensatz zur.ckgegeben wird, wenn kein dem Suchschl.ssel genau entsprechender Satz gefunden worden ist.

Wenn SET SOFTSEEK auf ON eingestellt und kein passender Satz f.r einen SEEK-Vorgang gefunden worden ist, wird der Datensatzzeiger auf den n.chsten Datensatz mit einem h.heren Schl.sselwert als das angegebene Suchargument positioniert. Durch SET FILTER und/oder SET DELETED nicht sichtbare Datens.tze werden bei der Suche nach dem n.chsth.heren Schl.sselwert .bersprungen. Wird kein Satz mit einem h.heren Schl.sselwert gefunden, wird der Datensatzzeiger auf LASTREC()+1 positioniert, EOF() gibt dann true (.T.) und FOUND() false (.F.) zur.ck. FOUND() gibt nur dann true (.T.), wenn der gesuchte Schl.sselwert gefunden worden ist. Ein mit dem n.chsth.heren Schl.sselwert gefundener Datensatz ergibt f.r FOUND() niemals true (.T.).

Wenn SET SOFTSEEK auf OFF eingestellt und kein passender Satz f.r einen SEEK-Vorgang gefunden worden ist, wird der Datensatzzeiger auf LASTREC()+1 positioniert, EOF() gibt true (.T.) und FOUND() false (.F.) zur.ck.

Hinweise

. SET RELATION: SET RELATION ignoriert SOFTSEEK bei der Aktualisierung des Datensatzzeigers in den abh.ngigen Arbeitsbereichen und verh.lt sich immer so, als ob SET SOFTSEEK auf OFF eingestellt ist.

Beispiele

Das folgende Beispiel zeigt m.gliche Ergebnisse eines Suchvorgangs mit SET SOFTSEEK ON:

```
SET SOFTSEEK ON
USE Vertretr INDEX Vertretr NEW
ACCEPT "Geben Sie den Vertreter ein: " TO cSuchen
SEEK cSuchen
DO CASE
CASE FIELD->Vertreter = cSuchen
  ? "Passenden Satz gefunden:", FOUND(), EOF(), FIELD->Vertreter
CASE .NOT. EOF()
  ? "N.chsth.heren Satz gefunden:", FOUND(), EOF(),;
  FIELD->Vertreter
OTHERWISE
  ? "Keinen passenden Satz gefunden:", FOUND(), EOF(),;
  FIELD->Vertreter
ENDCASE
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET TYPEAHEAD Legt die Gr..e des Tastatur-Eingabepuffers fest

---

Syntax

SET TYPEAHEAD TO <nTastenEingaben>

Argumente

<nTastenEingaben> bestimmt die Anzahl von Tasteneingaben, die der Tastatur-Eingabepuffer aufnehmen kann. <nTastenEingaben> kann von 0 bis 4096 reichen. Die Vorgabegr..e h.ngt vom jeweils eingesetzten Computermodell ab, das Minimum betr.gt aber immer 16.

Anwendung

SET TYPEAHEAD bestimmt die Gr..e des Tastaturpuffers von Clipper, der zur Zwischenspeicherung von Benutzereingaben dient. SET TYPEAHEAD begrenzt allerdings nicht die m.gliche Anzahl von mit KEYBOARD in den Tastaturpuffer abzulegenden Zeichen. Nach dem Aufruf von SET TYPEAHEAD wird zuerst der Tastaturpuffer gel.scht und dann auf die Gr..e <nTastenEingaben> eingestellt.

Die Einstellung von SET TYPEAHEAD TO 0 zeigt einen Spezialfall, bei dem eine Zwischenspeicherung von Tasteneingaben nicht erfolgt. Dies geschieht, indem Clipper keine ausstehenden Tasteneingaben vom BIOS-Tastaturpuffer abfragt und in den Clipper-Tastaturpuffer ablegt, au.er nat.rlich bei ausdr.cklichen Eingabeanforderungen. Das ist auch der Grund daf.r, weshalb spezielle Systemtasten wie [Alt]+[C] (Programmabbruch) und [Alt]+[D] (Aktivierung des Debuggers) nicht benutzt werden k.nnen, bis eine ausdr.ckliche Eingabeanforderung vorliegt. NEXTKEY() ist hiervon genauso betroffen, da diese Funktion nicht den n.chsten Tastencode zur.ckgeben kann, ohne ihn aus dem BIOS-Tastaturpuffer zu entfernen. NEXTKEY() gibt in diesem Fall einen Wert ungleich Null zur.ck, der aber nicht dem tats.chlichen n.chsten Tastencode entspricht.

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET UNIQUE* Schaltet die Ber.cksichtigung nicht eindeutiger Schl.ssel bei der Indizierung ein/aus

---

Syntax

SET UNIQUE on | OFF | <xlSchalter>

Argumente

ON bewirkt die Erzeugung von Indizes mit nur jeweils einmal vorkommenden, eindeutigen Schl.sselwerten.

OFF bewirkt die Erzeugung von Indizes mit mehreren gleichen, nicht unbedingt eindeutigen Schl.sselwerten.

<xlSchalter> ist ein logischer Ausdruck, der eingeklammert werden mu.. Dabei hat true (.T.) die gleiche Bedeutung wie ON und false (.F.) entspricht OFF.

Anwendung

SET UNIQUE ist ein Datenbankbefehl, der das Aufbauverhalten einer Indexdatei bestimmt. Die Einstellung von SET UNIQUE ON entspricht dem Befehl INDEX...UNIQUE und hat zur Folge, da. in die Indexdatei nur der jeweils erste von mehreren gleichen Schl.sselwerten mit aufgenommen wird. Diese Schl.sselwerte sind somit eindeutig, es kann in der Indexdatei keinen doppelten Schl.sselwert geben.

Dieses Verhalten beh.lt der so erzeugte Index f.r alle Operationen bei, wie z.B. bei einer Index-Aktualisierung und -Reindizierung oder bei Ausf.hrung des PACK-Befehles. Auch nachfolgendes Ein- und Ausschalten von SET UNIQUE ver.ndert das Verhalten dieses Indexes nicht mehr.

Die Ver.nderung von Schl.sselwerten in UNIQUE-Indexdateien kann wesentliche Folgen nach sich ziehen. Zum einen f.llt ein Schl.sselwert, der auf einen anderen, im Index schon existierenden Wert ge.ndert wird, aus dem Index heraus. Zum anderen gibt es keine automatische Aktualisierung, wenn der erste von mehreren gleichen Schl.sselwerten in einer Datenbank in den Index aufgenommen worden ist und in einen bisher nicht vorhandenen neuen Schl.sselwert ver.ndert wird. Hierbei wird zwar der neue Schl.sselwert im Index neu eingeordnet, nicht aber automatisch ein anderer Datensatz mit dem mehrfach vorkommenden alten Schl.sselwert in den Index hineingenommen. Der alte Schl.sselwert ist, obwohl in der zugrunde liegenden Datenbank mehrfach vorhanden, dann so lange nicht mehr im Index sichtbar, bis ein REINDEX, PACK oder INDEX...UNIQUE ausgef.hrt wird.

Die Einstellung von SET UNIQUE OFF hat zur Folge, da. in die Indexdatei alle vorkommenden Schl.sselwerte hintereinander mit aufgenommen werden. Auch hier hat nachfolgendes Ein- und Ausschalten von SET UNIQUE keinen Einflu. mehr auf das Verhalten dieses Indexes bei Aktualisierungen. Wird der Index komplett neu erzeugt, beeinflu.t die Einstellung von SET UNIQUE nat.rlich wieder die Auswahl der mit aufgenommenen Schl.sselwerte.

SET UNIQUE ist ein Kompatibilit.ts-Befehl und wird daher nicht zum Einsatz in neuen Applikationen empfohlen. Er wird durch den UNIQUE-Zusatz beim INDEX-Befehl ersetzt.

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SET WRAP Schaltet das Umlaufen des Cursors in Men.s ein/aus

Syntax

SET WRAP on | OFF | <xlSchalter>

Argumente

ON aktiviert das Umlaufen bei der Steuerung der Markierung in einem Light-bar-Men..

OFF deaktiviert das Umlaufen bei der Steuerung der Markierung in einem Light-bar-Men..

<xlSchalter> ist ein logischer Ausdruck, der eingeklammert werden mu.. Dabei hat true (.T.) die gleiche Bedeutung wie ON und false (.F.) entspricht OFF.

Anwendung

SET WRAP ist ein Men.befehl, der das Hin.berspringen der Markierung in einem @...PROMPT-Men. vom ersten zum letzten Men.punkt und zur.ck ein- bzw. ausschaltet. Wenn SET WRAP auf ON gestellt ist, wird eine auf dem letzten Men.punkt stehende Markierung auf den ersten Men.punkt bewegt, wenn [.] oder [.] gedr.ckt werden. Entsprechend wird die Markierung vom ersten auf den letzten Men.punkt bewegt, wenn [.] oder [.] gedr.ckt werden.

Wenn SET WRAP auf OFF eingestellt ist, bewirkt das Dr.cken der o.g. Tasten in der jeweiligen Konstellation keine Ver.nderung.

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SKIP Setzt den Datensatzzeiger auf eine neue Position

Syntax

SKIP [<nDatens.tze>] [ALIAS <idAlias> | <nArbeitsBereich>]

Argumente

<nDatens.tze> ist ein numerischer Ausdruck, der die Anzahl von Datens.tzen angibt, um die der Datensatzzeiger von der aktuellen Position zu bewegen ist. Positive Werte bewegen den Satzzeiger vorw.rts in Richtung Dateiende, negative Werte entsprechend r.ckw.rts in Richtung Dateianfang.

ALIAS <idAlias> <nArbeitsBereich> gibt den Alias-Namen als Literal-Bezeichner oder den Arbeitsbereich als numerischen Ausdruck an.

SKIP ohne Argument bewegt den Datensatzzeiger um einen Satz vorw.rts.

Anwendung

SKIP bewegt den Datensatzzeiger im aktuellen Arbeitsbereich unter Ber.cksichtigung der aktuellen Filterbedingung auf eine neue Position, die relativ zur bisherigen Position angegeben ist. SKIP wird normalerweise f.r Operationen verwendet, die zum n.chsten Datensatz einer Datenbank gehen m.ssen, z.B. Reportausgaben.

Mit der Angabe der Alias-Klausel kann der Datensatzzeiger des angegebenen Arbeitsbereiches bewegt werden, ohne da. der Arbeitsbereich explizit selektiert werden mu.te. SKIP kann vorw.rts oder r.ckw.rts ausgef.hrt werden. Ist kein kontrollierender Index aktiv, wird der Datensatzzeiger relativ zur aktuellen Position in der Ziel-Datenbank bewegt. Bei einem kontrollierenden Index wird der Datensatzzeiger relativ zur aktuellen Position im Index anstatt in der Ziel-Datenbank bewegt.

Ein SKIP-Befehl .ber das Dateiende hinaus setzt den Datensatzzeiger auf LASTREC()+1, und EOF() gibt true (.T.) zur.ck. Ein SKIP-Befehl .ber den Dateianfang hinaus setzt den Datensatzzeiger auf den ersten Satz und BOF() gibt true (.T.) zur.ck.

Jede .nderung des aktuellen, f.r andere Benutzer sichtbaren Arbeitsbereiches durch eine Datensatzbewegung (z.B. SKIP) zusammen mit RLOCK() wird in einer Netzwerkumgebung f.r die anderen Benutzer ebenfalls sichtbar, wenn die aktuelle Datenbank im shared-Modus ge.ffnet ist. Eine f.r andere Netzwerkapplikationen sichtbare Aktualisierung ohne Ver.nderung des Datensatzzeigers erreichen Sie mit SKIP 0. Wurden die .nderungen allerdings w.hrend einer Dateisperre (FLOCK()) durchgef.hrt, kann die Sichtbarkeit f.r andere Netzwerkstationen nur dann garantiert werden, wenn die Sperre aufgehoben, COMMIT ausgef.hrt und die Datei geschlossen worden ist. Weitere Informationen zur Programmierung von Netzwerkanwendungen finden Sie im Handbuch Programmierung und Hilfsprogramme.

Beispiele

. Das folgende Beispiel benutzt SKIP mit verschiedenen Argumenten und zeigt die Ergebnisse:

```
USE Kunden NEW
SKIP
? RECNO()      // Ergebnis: 2
SKIP 10
? RECNO()      // Ergebnis: 12
SKIP -5
? RECNO()      // Ergebnis: 7
```

. Dieses Beispiel bewegt den Datensatzzeiger in einem nicht selektierten Arbeitsbereich:

```
USE Kunden NEW
USE Rechnung NEW
SKIP ALIAS Kunden
```

. Das folgende Beispiel druckt einen Bericht, wobei der Datensatzzeiger die ganze Datenbank Verkauf.DBF mit SKIP sequentiell durchl.uft:

```
LOCAL nSeite := 0, nZeile := 99
USE Verkauf INDEX Verkauf NEW
SET PRINTER ON
SET CONSOLE OFF
DO WHILE .NOT. EOF()
  IF nZeile > 55
    IF nSeite <> 0
       EJECT
    ENDIF
    ? PADR("Seite", LTRIM(STR(nSeite++)), 72)
    ?? DTOC(DATE())
    ?
    ?
    ? PADC("Verkaufs.bersicht", 79)
    ?
    nZeile := 5
  ENDIF
  ? Name, Adresse, PLZ, PADR(RTRIM(Stadt), 20)
  nZeile++
  SKIP
ENDDO
SET CONSOLE ON
SET PRINTER OFF
CLOSE Verkauf
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

AFILL() F.llt die Elemente eines Arrays mit den angegebenen Werten

---

Syntax

AFILL(<aArray>, <expF.ll>, [<nStart>], [<nAnzahl>]) -> aArray

Parameter

<aArray> ist das zu f.llende Array.

<expF.ll> ist der Wert, mit dem jedes Element des Arrays gef.llt werden soll. Dies kann ein Ausdruck eines beliebigen, g.ltigen Datentyps sein.

<nStart> ist die Position des ersten zu f.llenden Elements. Entf.llt der Parameter, so ist der Vorgabewert das erste Element.

<nAnzahl> ist die Anzahl der Elemente, die ab <nStart> mit dem Wert aus <expF.ll> zu f.llen sind. Wird der Parameter nicht angegeben, so wird bis zum letzten Element gef.llt.

R.ckgabe

AFILL() liefert eine Referenz auf <aArray>.

Anwendung

AFILL() ist eine Arrayfunktion und f.llt Elemente eines Arrays mit dem Wert eines .ber <expF.ll> angegebenen beliebigen Datentyps, einschlie.lich Array, Codeblock oder NIL.

Warnung

AFILL() kann nicht dazu benutzt werden, um mehrdimensionale Arrays zu f.llen. In Clipper sind mehrdimensionale Arrays durch Verschachtelungen mit weiteren Arrays implementiert. Bei mehrdimensionalen Arrays zu f.llen, so werden Subarrays .berschrieben.

Beispiele

In diesem Beispiel wird ein Array mit drei Elementen erzeugt, allen Elementen der Wert .F. zugewiesen und danach den letzten beiden Elementen ein neuer Wert .T.:

```
LOCAL aLogisch[3]           // aLogisch = {NIL, NIL, NIL}
AFILL(aLogisch, .F.)        // aLogisch = {.F., .F., .F.}
AFILL(aLogisch, .T., 2,2)   // aLogisch = {.F., .T., .T.}
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

SORT Kopiert Datens.tze in sortierter Folge in eine neue .DBF-Datei

---

Syntax

SORT TO <xcDatenbank> ON <idFeld> [/[A | D] [C]]
[,<idFeld2> [/[A | D] [C]]...]
[<Wirkungsbereich>] [WHILE <lBedingung>] [FOR <lBedingung>]

Argumente

TO <xcDatenbank> ist der Name der Zieldatei f.r die sortierten Datens.tze. Der Name kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Falls keine andere Erweiterung angegeben ist, wird die Standard-Erweiterung .DBF hinzugef.gt.

ON <idFeld> gibt den Sortierschl.ssel an und mu. ein Datenfeld sein.

/[A | D] [C] bestimmen die Sortierfolge der <xcDatenbank>. /A sortiert in aufsteigender Folge. /D sortiert in absteigender Folge. /C ignoriert Gro.-/Kleinschreibung, d.h., es wird nicht zwischen Gro.- und Kleinbuchstaben unterschieden. Falls keine andere Sortierfolge angegeben ist, wird die Sortierfolge aufsteigend verwendet.

<Wirkungsbereich> spezifiziert den Teil, der in der aktuellen Datenbank sortiert werden soll. Die Vorgabe umfa.t alle Datens.tze (ALL).

WHILE <lBedingung> spezifiziert die Gruppe der Datens.tze, die sich vom aktuellen bis zum ersten, die Bedingung nicht erf.llenden Datensatz erstreckt.

FOR <lBedingung> selektiert alle Datens.tze innerhalb des <Wirkungsbereiches>, welche die angef.hrte Bedingung erf.llen und somit sortiert werden.

Anwendung

SORT ist ein Datenbankbefehl, der Datens.tze des aktuellen Arbeitsbereich in eine andere Datenbank in sortierter Folge kopiert. Clipper sortiert Zeichenfelder entsprechend des ASCII-Wertes jedes einzelnen Zeichens, au.er Sie haben die Klausel /C angegeben. Numerische Ausdr.cke werden entsprechend ihres Wertes und Datumsausdr.cke in chronologischer Folge sortiert, wobei ein leeres Datum immer am Dateianfang eingeordnet wird. Bei logischen Werten wird false (.F.) am Dateianfang und true (.T.) am Dateiende eingeordnet. Memo-Felder k.nnen nicht sortiert werden.

SORT versucht, den Sortiervorgang soweit wie m.glich im Speicher auszuf.hren und legt das Ergebnis in einer tempor.ren Datei auf Platte ab. Diese Datei kann so gro. werden wie die originale Datenbank. SORT verbraucht drei File Handles: je eins f.r die Quell-Datenbank, die Ziel-Datenbank und die tempor.re Datei.

In einer Netzwerkumgebung mu. die zu sortierende Datei mit FLOCK() gesperrt oder mit USE EXCLUSIVE ge.ffnet worden sein.

Hinweise

. Gel.schte S.tze in der Quelldatei: Wenn SET DELETED auf OFF eingestellt ist, werden alle gel.schten Datens.tze in die Ziel-Datenbank unter Beibehaltung bzw. .bernahme ihrer L.schmarken kopiert. Wenn SET DELETED auf ON eingestellt ist, werden zum L.schen markierte Datens.tze nicht in die Ziel-Datenbank .bernommen.

. SET EXACT: SET EXACT beeinflu.t SORT nicht.

Beispiele

Dieses Beispiel kopiert einen Teil einer Adressenliste in sortierter Folge in eine kleinere Liste zwecks nachfolgendem Ausdruck:

```
USE Mailing INDEX PLZ
SEEK "5000"
```

```
SORT ON Nachname, Vorname TO Einladg WHILE PLZ = "5000"
USE Einladg NEW
REPORT FORM EinListe TO PRINTER
```

Dateien

```
Bibliothek ist CLIPPER.LIB
```

See Also:

STORE* Weist einer oder mehreren Variablen einen Wert zu

Syntax

STORE <Ausdruck> TO <idVarListe>
<idVar> = <Ausdruck>
<idVar> := [<idVar2> := ...] <Ausdruck>

Argumente

<Ausdruck> ist ein Wert beliebigen Datentyps, der den angegebenen Variablen zugewiesen werden soll.

TO <idVarListe> ist eine Liste einer oder mehrerer LOCAL-, STATIC-, PUBLIC-, PRIVATE- oder FIELD-Variablen, denen Werte zugewiesen werden sollen. Falls eine der angegebenen Variablen nicht sichtbar ist oder nicht existiert, wird sie als PRIVATE-Variable angelegt und ihr wird <Ausdruck> zugewiesen.

Anwendung

STORE weist einer oder mehreren Variablen beliebiger Speicherklassen einen Wert zu. Clipper kennt die Speicherklassen LOCAL, STATIC, PUBLIC, PRIVATE und FIELD.
Anders als beim Befehl REPLACE, wo nur eine FIELD-Variable einen Wert erhalten kann, erlaubt es der STORE-Befehl, jedem beliebigen Variablentyp einen Wert zuzuordnen. Unklare, d.h. zur Compilierzeit nicht deklarierte oder nicht .ber einen Alias-Verweis eingeordnete Namen in <idVarListe> werden als MEMVAR angenommen. Zur Laufzeit wird dann eine PRIVATE-Variable angelegt, wenn keine PRIVATE- oder PUBLIC-Variable mit dem entsprechenden Namen existiert.

Besteht ein Konflikt zwischen mehreren Variablen gleichen Namens, hat die jeweils zuletzt deklarierte Variable Vorrang. Ist die Variable nicht deklariert, wird der Wert <Ausdruck> erst einer FIELD-, dann einer PRIVATE- und schlie.lich einer PUBLIC-Variablen zugewiesen.

Eine Deklaration kann ge.ndert werden, indem Sie <idVar> mit einem Alias-Verweis angeben. Dies ist bei Vorliegen einer Datenbankfeld-Variablen der Name des jeweiligen Arbeitsbereiches. PRIVATE- und PUBLIC-Variablen k.nnen .ber den Alias-Verweis f.r Speichervariablen (MEMVAR->) gekennzeichnet werden. Felder im aktuellen Arbeitsbereich k.nnen im Gegensatz zu nicht selektierten Arbeitsbereichen .ber FIELD-> angesprochen werden.

Prinzipiell sollten alle Variablen au.er Datenbankfeldern deklariert werden. Feldvariablen sollten immer mit vorgestelltem Alias-Verweis angesprochen werden. Vom Einsatz von PRIVATE- und PUBLIC-Variablen sollte abgesehen werden, da sie die Grunds.tze modularer Programmierung mi.achten und wesentlich langsamer als LOCAL- und STATIC-Variablen sind.

Achtung

STORE geht nicht nur von Speichervariablen als Ziel der Wertzuweisungen aus. Wenn eine Variable als FIELD deklariert wird, ersetzt STORE das entsprechende Feld in einer Datenbank.

STORE entspricht den einfachen Zuweisungsoperatoren = und :=.
Tats.chlich wird STORE auch intern vom Pre-Prozessor in eine Anweisung mit dem in-line-Operator := umgewandelt. STORE ist ein Kompatibilit.ts-Befehl und wird daher nicht zum Einsatz in neuen Applikationen empfohlen. Clipper 5.0 stellt verschiedene, m.chtige Zuweisungsoperatoren zur Verf.gung, die STORE ersetzen. Dazu geh.ren :=, ++, --, +=, -=, *= und /=. Eine genaue Erl.uterung der Operatoren- und Variablentypen finden Sie in diesem Handbuch im Kapitel Grundkonzepte.

Hinweise

. Zuweisung eines Wertes an ein komplettes Array: Ein einzelner Wert kann einem kompletten Array in Clipper 5.0 nur .ber die Funktion AFILL() zugewiesen werden. Keiner der Zuweisungsoperatoren oder STORE sind hierf.r verwendbar.

. Memo-Felder: Wird einer Variablen ein Memo-Feld zugewiesen, enth.lt die Zielvariable danach einen Zeichenwert.

Beispiele

. Das folgende Beispiel erzeugt nicht deklarierte PRIVATE-Variablen und weist ihnen Werte zu:

```
STORE "Zeichenkette" TO cVar1, cVar2, cVar3
cVar1 = "String2"
cVar2 = MEMVAR->cVar1
```

. verwendet wird. Beide Methoden produzieren einen identischen Code.

```
STORE "Zeichenkette" TO cVar1, cVar2, cVar3
cVar1 := cVar2 := cVar3 := "Zeichenkette"
```

. Dieses Beispiel weist ein und demselben Feld einen Wert zu, wobei das Feld explizit .ber einen Alias-Verweis angesprochen wird. Dieser ist einmal der Feldverweis FIELD. und einmal der tats.chliche Alias-Name des aktuellen Arbeitsbereiches.

```
USE Verkauf NEW
FIELD->KundSaldo = 1.200,99
Verkauf->KundSaldo = 1.200,99
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

SUM Summiert numerische Ausdr.cke und ordnet die Ergebnisse Variablen zu

---

Syntax

SUM <AusdrucksListe> TO <idVarListe>
[<Wirkungsbereich>] [WHILE <lBedingung>] [FOR <lBedingung>]

Argumente

<AusdrucksListe> ist eine Liste von numerischen Werten, die f.r jeden verarbeiteten Datensatz summiert werden sollen.

<idVarListe> ist eine Liste von Variablen, denen die Ergebnisse der Summierung zugewiesen werden sollen. Falls eine der angegebenen Variablen nicht sichtbar ist oder nicht existiert, wird sie als PRIVATE-Variable angelegt. <idVarListe> mu. die gleiche Anzahl an Elementen enthalten wie <AusdrucksListe>.

<Wirkungsbereich> spezifiziert den Teil, der in der aktuellen Datenbank summiert werden soll. Die Vorgabe umfa.t alle Datens.tze (ALL).

WHILE <lBedingung> spezifiziert die Gruppe der Datens.tze, die sich vom aktuellen bis zum ersten, die Bedingung nicht erf.llenden Datensatz erstreckt.

FOR <lBedingung> selektiert alle Datens.tze innerhalb des <Wirkungsbereiches>, welche die angef.hrte Bedingung erf.llen und somit summiert werden.

Anwendung

SUM ist ein Datenbankbefehl, der eine Gruppe von numerischen Ausdr.cken f.r eine Anzahl von S.tzen im aktuellen Arbeitsbereich aufsummiert und die Ergebnisse entsprechenden Variablen zuweist. Die Variablen in <idVarListe> k.nnen vom Typ LOCAL, STATIC, PUBLIC, PRIVATE oder FIELD sein.

Die Angabe von <AusdrucksListe> ist zwingend notwendig und nicht optional wie in anderen dBASE-Dialekten.

Beispiele

Das folgende Beispiel zeigt den Einsatz von SUM:

```
LOCAL nGesamtPreis, nSteuer
USE Verkauf NEW
SUM Preis * .14, Betrag TO nSteuer, nGesamtPreis
//
? nGesamtPreis    // Ergebnis: 1.000,00
? nSteuer         // Ergebnis:   140,00
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

TEXT* Gibt einen Textblock aus

Syntax

```
TEXT [TO PRINTER] [TO FILE <xcDatei>]
  <Text>...
ENDTEXT
```

Argumente

<Text> ist der Textblock, der auszugeben ist. Der Text wird genauso formatiert dargestellt, wie er eingegeben worden ist.

TO PRINTER sendet die Ausgabe zus.tzlich zum Drucker.

TO FILE <xcDatei> leitet die Ausgabe zus.tzlich in die Datei <xcDatei> um. <xcDatei> kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Falls keine andere Erweiterung angegeben ist, wird die Standard-Erweiterung .TXT hinzugef.gt.

Anwendung

Der Textblock wird genauso ausgegeben, wie er in die Programmdatei eingegeben worden ist, also auch inklusive Einr.ckungen. Harte Carriage Returns (ASCII 13) werden als neue Zeile, weiche Carriage Returns als ASCII-Zeichen CHR(141) ausgegeben. Makro-Variablen innerhalb des Textblockes werden expandiert, Makro-Ausdr.cke werden allerdings nicht verarbeitet.

TEXT...ENDTEXT ist ein Kompatibilit.ts-Befehl und wird daher nicht zum Einsatz in neuen Applikationen empfohlen. Clipper bietet bessere Mittel zur Textverarbeitung und -ausgabe. Sie k.nnen z.B. MEMOLINE() in Kombination mit MLCOUNT() einsetzen, um den Umbruch von langen Zeichenketten unter Ber.cksichtigung einer angegebenen Zeilenl.nge zu erzeugen. ? oder @...SAY k.nnen formatierten Text, der mit MEMOLINE() aus einem l.ngeren String herausgenommen worden ist, darstellen.

Beispiele

Das folgende Beispiel zeigt, wie Sie mit TEXT...ENDTEXT einen Serienbrief drucken k.nnen:

```
USE Verkauf NEW
DO WHILE .NOT. EOF()
  SerienBrief()
  SKIP
ENDDO
RETURN
FUNCTION SerienBrief
  LOCAL dDatum := DTOC(DATE()), cVerkaeufer := RTRIM(Verkaeufer)
  TEXT TO PRINTER

  &dDatum

  Sehr geehrter &cVerkaeufer,
  wie geht es Ihnen?
  ENDTEXT
  EJECT
RETURN NIL
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

TOTAL Fa.t Datens.tze anhand eines Schl.sselausdrucks zusammen und summiert diese in eine neue .DBF-Datei

---

Syntax

TOTAL ON <Schl.sselAusdruck> [FIELDS <idFelderListe>] TO <xcDatenbank> [<Wirkungsbereich>] [WHILE <lBedingung>] [FOR <lBedingung>]

Argumente

ON <Schl.sselausdruck> definiert eine Gruppe von Datens.tzen, die in der Zieldatei insgesamt einen neuen Datensatz ergeben sollen. Um die Zusammenfassung korrekt auszuf.hren, sollte die Quelldatei auf diesen <Schl.sselausdruck> indiziert oder sortiert sein.

FIELDS <idFelderListe> gibt eine Liste der numerischen Felder an, die mit TOTAL summiert werden sollen. Wenn die FIELDS-Klausel nicht angegeben ist, werden keine numerischen Felder summiert. Stattdessen enth.lt jedes numerische Feld in der Zieldatei den Wert des jeweils ersten Datensatzes in der Quelldatei, der den Schl.sselausdruck erf.llt.

TO <xcDatenbank> gibt den Namen der Zieldatei an, in welche die zusammengefa.ten Datens.tze kopiert werden sollen. Der Name kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Falls keine andere Erweiterung angegeben ist, wird die Standard-Erweiterung .DBF hinzugef.gt.

<Wirkungsbereich> spezifiziert den Teil, der in der aktuellen Datenbank zusammengefa.t werden soll. Die Vorgabe umfa.t alle Datens.tze (ALL).

WHILE <lBedingung> spezifiziert die Gruppe der Datens.tze, die sich vom aktuellen bis zum ersten, die Bedingung nicht erf.llenden Datensatz erstreckt.

FOR <lBedingung> selektiert alle Datens.tze innerhalb des <Wirkungsbereiches>, welche die angef.hrte Bedingung erf.llen und somit zusammengefa.t werden.

Anwendung

TOTAL ist ein Datenbankbefehl, der die aktuelle Datenbank sequentiell verarbeitet und dabei die Datens.tze anhand des spezifizierten Schl.sselwertes zusammenfa.t und in eine neue Datenbank schreibt. TOTAL kopiert zuerst die Struktur der aktuellen Datenbank mit Ausnahme der Memo-Felder in die <xcDatenbank>. Danach werden die innerhalb des angegebenen Wirkungsbereiches liegenden Datens.tze der aktuellen Datenbank Satz f.r Satz verarbeitet. Jeder Datensatz mit einem neuen <Schl.sselausdruck> wird in die neue Zieldatei kopiert. Die Werte von den, in der <idFelderListe> aufgef.hrten numerischen Feldern werden bei aufeinanderfolgenden Datens.tzen mit demselben <Schl.sselausdruck> aufaddiert, die Summen werden in den korrespondierenden Feldern der neuen Datenbank abgelegt. Die Zusammenfassung bzw. Addition erfolgt, bis ein Datensatz mit einem neuen Schl.sselwert entdeckt wird. F.r diesen Schl.sselwert wird der Vorgang bis zum n.chsten neuen Schl.sselwert wiederholt usw.

Aufgrund der beschriebenen sequentiellen Verarbeitung mu. die Quell-Datenbank auf den <Schl.sselausdruck> indiziert oder sortiert sein, damit alle Datens.tze korrekt zusammengefa.t werden k.nnen.

Bei der Addition von numerischen Feldern ist darauf zu achten, da. die Felder der Quelldatei gro. genug sind, um die gr..tm.gliche Summe f.r dieses Feld aufnehmen zu k.nnen. .bersteigt die Summe eines numerischen Feldes die Anzahl der in der Felddefinition angegebenen Ziffern, tritt ein Laufzeitfehler auf.

Hinweise

. Gel.schte Datens.tze in der Quelldatei: In der Quelldatei als gel.scht markierte Datens.tze werden von TOTAL mitber.cksichtigt, es sei denn, SET DELETED ist auf ON eingestellt. Die L.schmarkierungen werden

allerdings nicht in die neue Datenbank kopiert.

Beispiele

In diesem Beispiel wird eine Datenbank anhand des Schl.sselausdrucks der kontrollierenden Indexdatei zusammengefa.t. Da hier der Index-Schl.sselausdruck .ber einen Makro-Ausdruck ermittelt wird, erfolgt zuerst die Auswertung des Makros. Erst danach ersetzt die daraus resultierende Zeichenkette den <Schl.sselausdruck> in TOTAL, und der TOTAL-Proze. beginnt.

```
USE Verkauf INDEX Filiale NEW
TOTAL ON &(INDEXKEY(0)) FIELDS Umsatz TO Summen
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

TYPE Gibt den Inhalt einer Textdatei aus

Syntax

TYPE <xcDatei> [TO PRINTER] [TO FILE <xcAusgabeDatei>]

Argumente

<xcDatei> ist der Name einer auszugebenden Datei. <xcDatei> kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Hier mu. eine Erweiterung angegeben werden, eine Standard-Erweiterung existiert nicht.

TO PRINTER sendet die Ausgabe zus.tzlich zum Drucker.

TO FILE <xcAusgabeDatei> leitet die Ausgabe zus.tzlich in die Datei <xcAusgabeDatei> um. <xcAusgabeDatei> kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Falls keine andere Erweiterung angegeben ist, wird die Standard-Erweiterung .TXT hinzugef.gt.

Anwendung

TYPE ist ein Konsolen-Befehl, der den Inhalt einer Textdatei auf dem Bildschirm und optional dem Drucker und/oder in eine Textdatei ausgibt. Durch Angabe von SET CONSOLE OFF l..t sich die Bildschirmausgabe verhindern, w.hrend der Textblock zum Drucker oder in eine Textdatei gesendet wird.

Wenn <xcDatei> ohne Pfad- oder Laufwerksbezeichner angegeben wird, sucht TYPE erst im mit SET DEFAULT eingestellten Vorgabe-Verzeichnis und danach im aktuellen Pfad. Wenn <xcAusgabeDatei> ohne Pfad- oder Laufwerksbezeichner angegeben wird, erzeugt TYPE die Datei im durch SET DEFAULT eingestellten Verzeichnis.

TYPE f.hrt keine besondere Formatierung des aus der Datei eingelesenen Textes durch. Eine .berschrift oder Seitennummerierung erfolgt w.hrend des Ausdruckes nicht.

Die Ausgabe kann mit [Strg]+ [S] unterbrochen werden. Die [Esc]-Taste kann hierf.r nicht verwendet werden (au.er durch eine Umdefinierung der Taste z.B. mit SET KEY).

Beispiele

Das folgende Beispiel benutzt den TYPE-Befehl:

```
TYPE Haupt.PRG TO PRINTER
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

UNLOCK Deaktiviert die vom aktuellen Anwender eingestellten Datei- und Datensatzsperren

---

Syntax

UNLOCK [ALL]

Argumente

ALL hebt alle aktuellen Sperren in allen Arbeitsbereichen auf. Wenn ALL nicht angegeben ist, wird nur die Sperre des aktuellen Arbeitsbereiches gel.scht.

Anwendung

UNLOCK ist ein Netzwerkbefehl, der die vom aktuellen Anwender eingestellten Datei- und Datensatzsperren ganz oder teilweise deaktiviert. UNLOCK wird dann benutzt, wenn Sie die aktuelle Sperre ohne Setzen einer neuen Sperre aufheben m.chten. Sowohl FLOCK() als auch RLOCK() l.schen die aktuelle Sperre, bevor sie eine neue Sperre aktivieren.

Nach Durchf.hrung von UNLOCK werden die f.r andere Netzwerkstationen sichtbaren .nderungen von Datenbank- oder Memo-Dateien allerdings nur dann garantiert auf Platte geschrieben, wenn COMMIT ausgef.hrt oder die Datei geschlossen worden ist.

Weitere Informationen zur Programmierung von Netzwerkanwendungen und zur Sichtbarkeit von Aktualisierungen finden Sie im Handbuch Programmierung und Hilfsprogramme.

Hinweise

. SET RELATION: UNLOCK hebt eine Datensatzsperre in einer SET RELATION-Kette so lange nicht auf, bis UNLOCK ALL ausgef.hrt worden ist.

Beispiele

. In diesem Beispiel wird eine Aktualisierung versucht, welche die Sperre des Datensatzes verlangt. Wenn RLOCK() erfolgreich ist, wird der Satz .ber eine UDF aktualisiert und anschlie.end UNLOCK zur Aufhebung der Sperre ausgef.hrt.

```
USE Verkauf INDEX Verkauf SHARED NEW
IF RLOCK()
  UpdateRecord()
  UNLOCK
ELSE
  ? "Datensatz konnte nicht aktualisiert werden!"
  BREAK
ENDIF
```

. Dieses Beispiel zeigt, wie PACK im Netzwerk ausgef.hrt werden kann. Wenn FLOCK() Erfolg hatte, wird PACK durchgef.hrt. Danach wird die Dateisperre mit UNLOCK aufgehoben.

```
USE Verkauf INDEX Verkauf SHARED NEW
IF FLOCK()
  PACK
  UNLOCK
ELSE
  ? "Datei konnte nicht 'gepackt' werden!"
  BREAK
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

UPDATE Aktualisiert die selektierte Datenbank anhand einer anderen Datenbank

---

Syntax

UPDATE FROM <xcAlias> ON <Schl.sselausdruck> [RANDOM]
REPLACE <idFeld> WITH <Ausdruck> [, <idFeld2> WITH <Ausdruck2>...]

Argumente

FROM <xcAlias> gibt den Alias-Namen des Quell-Arbeitsbereiches an, der f.r die Aktualisierung der selektierten Datenbank verwendet wird. Der Name kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden.

ON <Schl.sselausdruck> gibt einen Ausdruck an, der mit den entsprechenden Datens.tzen des FROM- und des aktuellen Arbeitsbereiches .bereinstimmen mu., damit sie aktualisiert werden.

REPLACE <idFeld> gibt ein Feld im aktuellen Arbeitsbereich an, das durch einen neuen Wert ersetzt werden soll.

WITH <Ausdruck> ist der Wert, durch den das zugeh.rige Feld ersetzt werden soll. Jedes Feld im FROM-Arbeitsbereich mu. mit dem korrekten Alias-Verweis aufgef.hrt werden.

RANDOM bewirkt, da. keine besondere Reihenfolge der Datens.tze in der FROM-Datenbank vorausgesetzt wird. Bei Angabe dieses Zusatzes mu. die selektierte Datenbank allerdings auf den <Schl.sselausdruck> indiziert sein.

Anwendung

UPDATE ist ein Datenbankbefehl, der Felder des aktuellen Arbeitsbereiches durch Werte eines anderen Arbeitsbereiches ersetzt, wobei die Verbindung der Datens.tze beider Arbeitsbereiche .ber den <Schl.sselausdruck> erfolgt. UPDATE kann nur Felder im aktuellen Arbeitsbereich ersetzen. Es erfolgt entweder eine eins-zu-eins- oder eins-zu-viele-Verbindung mit dem FROM-Arbeitsbereich. Das bedeutet, da. UPDATE im aktuellen Arbeitsbereich nur eindeutige Schl.sselwerte ersetzt. Ist ein Schl.sselwert mehrfach in der aktuellen Datenbank vorhanden, wird nur der zuerst vorkommende Datensatz durch UPDATE aktualisiert. Im Quell-Arbeitsbereich (FROM) k.nnen aber beliebig viele gleiche Schl.sselwerte vorkommen.

Abh.ngig von der Indizierung oder Sortierung des aktuellen Arbeitsbereiches nach dem <Schl.sselausdruck>, kann UPDATE mit dem Zusatz RANDOM verwendet werden oder nicht. Wenn RANDOM nicht angegeben wird, m.ssen sowohl der Quell- als auch der Ziel-Arbeitsbereich nach <Schl.sselausdruck> indiziert oder sortiert sein. Im Gegensatz dazu mu. bei Angabe des Zusatzes RANDOM nur der aktuelle Arbeitsbereich indiziert oder sortiert sein. Die Datens.tze des Quell-Arbeitsbereiches d.rfen dann in jeder beliebigen Reihenfolge vorliegen.

Beim Einsatz von UPDATE in einer Netzwerkumgebung mu. die aktuelle Datenbank mit FLOCK() oder USE EXCLUSIVE gesperrt sein. Die Quell-Datenbank kann in jedem Modus benutzt werden. Weitere Informationen zur Programmierung von Netzwerkanwendungen finden Sie im Handbuch Programmierung und Hilfsprogramme.

Hinweise

. Gel.schte Datens.tze: Wenn SET DELETED auf ON eingestellt ist, werden alle (also auch die gel.schten) Datens.tze der Ziel-Datenbank aktualisiert. In der Quelldatei werden als gel.scht markierte Datens.tze allerdings ignoriert.

Beispiele

Das folgende Beispiel aktualisiert die Kundendatei mit ausstehenden Rechnungsbetr.gen:

```
  USE Rechnung NEW
  USE Kunden INDEX Kunden NEW
  UPDATE FROM Rechnung ON Zuletzt;
    REPLACE Offen WITH Offen + Rechnung->Umsatz RANDOM
```

Dateien

```
  Bibliothek ist CLIPPER.LIB
```

See Also:

USE .ffnet eine existierende .DBF-Datei und damit verbundene Dateien

---

Syntax

```
USE [<xcDatenbank>
  [INDEX <xcIndexListe>]
  [ALIAS <idAlias>]
  [EXCLUSIVE | SHARED]
  [NEW] [READONLY]]
```

Argumente

<xcDatenbank> ist der Name der zu .ffnenden Datenbank. Der Name kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden.

INDEX <xcIndexListe> ist eine Liste mit bis zu 15 durch Kommata voneinander getrennten Namen von Indexdateien (.NTX oder .NDX), die im aktuellen Arbeitsbereich ge.ffnet werden sollen. Ein Dateiname kann als Literal oder eingeklammerter Zeichenausdruck angef.hrt werden. Der erste Index in der Liste wird zum kontrollierenden Index. Jeder Name, der einen Nullstring ("") ergibt oder leer ist, wird ignoriert.

ALIAS <idAlias> ist ein Name, der mit dem Arbeitsbereich zu assoziieren ist, wenn die Datenbank ge.ffnet wird. Wird der ALIAS-Zusatz nicht angegeben, wird der Name der ge.ffneten Datenbank als Alias-Name vorgegeben und bei Alias-Verweisen verwendet.

EXCLUSIVE .ffnet die Datei in einer Netzwerkumgebung im non-shared-Modus. In diesem Fall kann kein anderer Benutzer die Datenbank .ffnen, bis sie geschlossen worden ist.

SHARED .ffnet die Datei in einer Netzwerkumgebung im shared-Modus. Die Angabe dieser Klausel hebt eine augenblicklich vorhandene EXCLUSIVE-Einstellung auf.

NEW .ffnet die <xcDatenbank> im n.chsten verf.gbaren Arbeitsbereich, der danach zum aktuellen Arbeitsbereich wird. Wird NEW nicht angegeben, .ffnet USE die Datenbank im aktuellen Arbeitsbereich.

READONLY .ffnet die <xcDatenbank> im NUR LESEN-Modus. Auf diese Weise k.nnen Dateien, die als read-only gekennzeichnet sind, ge.ffnet werden. Ist eine .ffnung der <xcDatenbank> in diesem Modus nicht m.glich, wird ein Laufzeitfehler erzeugt. Wenn READONLY nicht angegeben wird, .ffnet USE die <xcDatenbank> im read-write-Modus.

Die Verwendung von USE ohne Argument schlie.t die im aktuellen Arbeitsbereich ge.ffnete Datenbank.

Anwendung

USE .ffnet eine existierende Datenbank (.DBF), automatisch eine evtl. zugeh.rige Memo-Datei (.DBT) und optional zugeh.rige Indexdateien (.NTX oder .NDX) im aktuellen oder n.chstverf.gbaren Arbeitsbereich. Clipper unterst.tzt f.r DOS 3.3 und h.here Versionen 250 Arbeitsbereiche mit maximal 255 ge.ffneten Dateien. Bevor USE die neue Datenbank mit den zugeh.rigen Dateien .ffnet, werden zuerst alle im aktuellen Arbeitsbereich bereits ge.ffneten Dateien geschlossen. Nach .ffnung einer Datenbank steht der Datensatzzeiger anf.nglich auf dem ersten logischen Datensatz (Satz 1, wenn keine Indexdatei angegeben worden ist).

In einer Netzwerkumgebung k.nnen Datenbanken entweder als EXCLUSIVE oder SHARED ge.ffnet werden. Die .ffnung im Modus EXCLUSIVE verweigert allen anderen Benutzern im Netzwerk die .ffnung dieser Datenbank, bis sie wieder geschlossen wird. SHARED hingegen erlaubt anderen Benutzern den gleichzeitigen Zugriff auf die Datenbank. In diesem Fall mu. der Programmierer in seinem Programm die notwendigen .berpr.fungen und Vorsichtsma.nahmen treffen, damit die Datenintegrit.t und -konsistenz gesichert ist. In Clipper gibt es daf.r die zwei Funktionen FLOCK() und RLOCK(), die anderen Benutzern den Zugriff auf einen Arbeitsbereich oder einen Datensatz verweigern. Wenn der USE-Befehl weder mit EXCLUSIVE noch

mit SHARED angegeben wird, erfolgt die .ffnung der Datenbank entsprechend der Einstellung von SET EXCLUSIVE. Sie sollten bei der Programmierung in Clipper 5.0 aber jede Datenbank unter Angabe von EXCLUSIVE oder SHARED .ffnen, da SET EXCLUSIVE ein Kompatibilit.ts-Befehl und wird daher nicht zum Einsatz in neuen Applikationen empfohlen ist.

Ein .ffnungsvorgang im Netzwerk erfordert gewisse zus.tzliche Aktionen, um erfolgreich durchgef.hrt zu werden. Als erstes mu. der USE-Befehl ohne die INDEX-Klausel angegeben werden. Der Erfolg des .ffnungsversuches wird danach mit NETERR() abgefragt. Gibt NETERR() false (.F.) zur.ck, ist die Datei.ffnung gelungen, und Sie k.nnen die Indexdateien mit SET INDEX TO <xcIndexListe> .ffnen. Bei erfolgloser .ffnung kann der .ffnungsversuch f.r eine gewisse Zeit wiederholt werden, um so auszuschlie.en, da. die zu .ffnende Datenbank nur kurzzeitig gesperrt war. Eine solche Sperre tritt nur dann in Kraft, wenn ein anderer Benutzer die entsprechende Datenbank im Netz mit USE EXCLUSIVE ge.ffnet hat. Weitere Informationen zur .ffnung von Dateien und Programmierung von Netzwerkanwendungen finden Sie im Handbuch Programmierung und Hilfsprogramme.

Indexdateien k.nnen entweder mit USE oder SET INDEX ge.ffnet werden. Der erste Index in der <xcIndexListe> gibt denjenigen Index an, nach dem alle Datens.tze geordnet werden, wenn auf sie zugegriffen wird. Dieser Index wird als kontrollierender Index bezeichnet. Der kontrollierende Index kann ohne das Schlie.en von Dateien mit dem Befehl SET ORDER ge.ndert werden.

Wenn Sie die Datenbank und ihre zugeh.rigen Dateien im aktuellen Arbeitsbereich schlie.en m.chten, verwenden Sie USE oder CLOSE ohne Argumente. Die Datenbanken in allen Arbeitsbereichen schlie.en Sie mit CLOSE DATABASES. Wenn Sie die Indexdateien im aktuellen Arbeitsbereich schlie.en m.chten, ohne die Datenbank selbst zu schlie.en, verwenden Sie CLOSE INDEX oder SET INDEX TO ohne Argumente.

Weitere Erl.uterungen zum Clipper-Datenbanksystem finden Sie in diesem Handbuch im Kapitel Grundkonzepte.

Hinweise

. F50 eingestellt sind, k.nnen in einer Clipper-Applikation maximal 50 Dateien gleichzeitig ge.ffnet sein. Der ideale Wert f.r den F-Parameter ist immer mindestens 5 gr..er als die FILES-Anzahl. Beide Werte sollten aufgrund eines bei verschiedenen DOS-Versionen enthaltenen internen Fehlers ungerade sein.

  Neben den beschriebenen Parametern beeinflu.t aber auch die verwendete Version des Betriebssystems die maximale Dateianzahl. Bei DOS-Versionen kleiner als 3.3 k.nnen maximal 20 Dateien gleichzeitig ge.ffnet sein. Bei DOS-Versionen ab 3.3 k.nnen maximal 255 Dateien (davon 250 unter Clipper) gleichzeitig ge.ffnet sein.

. .ffnen einer Datenbank in mehreren Arbeitsbereichen gleichzeitig: Obwohl ein und dieselbe Datenbank in einer Netzwerkumgebung in mehreren Arbeitsbereichen gleichzeitig ge.ffnet werden kann, ist diese Praxis alles andere als ratsam.

Beispiele

. Das folgende Beispiel demonstriert die .ffnung einer Datenbank mitsamt zugeh.riger Indexdateien in einer Netzwerkumgebung im shared-Modus. Wenn NETERR() false (.F.) zur.ckgibt, war die .ffnung erfolgreich und die Indexdateien k.nnen ge.ffnet werden.

```
USE Konten SHARED NEW
IF .NOT. NETERR()
  SET INDEX TO KontoNam, BLZ
ELSE
  ? "Datei konnte nicht ge.ffnet werden!"
  BREAK
```

```
ENDIF
```

. Dieses Beispiel zeigt, wie eine Datenbank mit mehreren, .ber
erweiterte Ausdr.cke angegebene Indexdateien ge.ffnet werden kann.
Erw.hnenswert ist hierbei au.erdem die Anlage des Arrays mit den
Indexdateinamen als Konstanten-Array.

```
xcDatenbank = "MeineDBF"
xcIndex = {"MeinIndex1", "MeinIndex2", "MeinIndex3"}
USE (xcDatenbank) INDEX (xcIndex[1]), (xcIndex[2]), (xcIndex[3])
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

AINS() Einf.gen eines Arrayelementes mit dem Wert NIL

Syntax

AINS(<aArray>, <nPosition>) -> aArray

Parameter

<aArray> ist das Array, in das ein neues Element eingef.gt wird.

<nPosition> ist die Position, an der das neue Element eingef.gt wird.

R.ckgabe

AINS() gibt eine Referenz auf <aArray> zur.ck.

Anwendung

AINS() ist eine Arrayfunktion, welche in einem Array ein neues Element einf.gt. Das neu eingef.gte Element ist vom Typ NIL, bis ein anderer Wert zugewiesen wird. Nachdem das neue Element eingef.gt ist, geht das letzte Element verloren, und die Elemente ab der Einf.geposition werden um ein Element nach unten verschoben.

Warnung

AINS() sollte bei mehrdimensionalen Arrays sehr vorsichtig angewendet werden. In Clipper sind mehrdimensionale Arrays durch Verschachtelung mit weiteren Subarrays implementiert. AINS() mit einem mehrdimensionalen Array verwirft das letzte Subarray in <aArray>, dadurch gehen eine oder mehrere Dimensionen verloren. Um eine neue Dimension in ein Array einzuf.gen, sollte erst mit AADD() oder ASIZE() ein neues Element an das Ende des Arrays angeh.ngt werden, bevor AINS() benutzt wird.

Beispiele

. Dieses Beispiel demonstriert den Effekt, wenn AINS() benutzt wird:

```
LOCAL aArray := { 1, 2, 3 }      // Ergibt { 1, 2, 3 }
AINS(aArray, 2)                  // Ergibt { 1, NIL, 2 }
```

. Ein Beispiel von AINS() unter Verwendung eines mehrdimensionalen Arrays:

```
LOCAL aArray := {{ 1, 1}, { 2, 2 }, { 3, 3 }}
AINS(aArray, 2)                  // Ergibt {{1, 1},NIL,{2, 2}}
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

WAIT* Unterbricht die Programmausf.hrung, bis eine Taste gedr.ckt worden ist

---

Syntax

WAIT [<PromptAusdruck>] [TO <idVar>]

Argumente

<PromptAusdruck> ist ein Ausdruck beliebigen Datentyps, der als Eingabeanforderung (Prompt) ausgegeben wird. Wird ein <PromptAusdruck> nicht angegeben, gibt WAIT den Vorgabetext "Press any key to continue" aus.

TO <idVar> gibt eine Variable aus einer beliebigen Speicherklasse an, der die Tastatureingabe als Zeichenwert zugewiesen werden soll. Wenn diese Variable nicht existiert oder nicht sichtbar ist, wird eine PRIVATE-Variable angelegt, der dann das eingegebene Zeichen zugewiesen wird.

Anwendung

WAIT ist ein Konsolen-Befehl und Wartezustand, der nach Ausgabe von Carriage Return/Line Feed eine Eingabeanforderung auf dem Bildschirm ausgibt. Wenn die Klausel TO angegeben wird, bekommt <idVar> die gedr.ckte Taste als Zeichenwert zugewiesen. Wurde die Taste [Alt]oder [Strg]gedr.ckt, weist WAIT der Variablen <idVar> den Wert CHR(0) zu. Sie k.nnen auch andere als alphanumerische Zeichen eingeben, indem Sie den entsprechenden Alt-Tastencode eingeben. Alle darstellbaren Zeichen k.nnen damit auf dem Bildschirm ausgegeben werden. Funktionstasten werden nicht ber.cksichtigt, au.er sie sind .ber SET KEY oder SET FUNCTION umdefiniert worden.

WAIT ist ein Kompatibilit.ts-Befehl und wird daher nicht zum Einsatz in neuen Applikationen empfohlen. Er wird durch @...GET/READ und INKEY() zur Abfrage von einzelnen Zeichen ersetzt.

Hinweise

. WAIT ohne Prompt: Sie k.nnen ein Programm ohne Ausgabe eines Prompts unterbrechen, wenn Sie entweder WAIT "" oder INKEY(0) verwenden. Letztere Funktion ist vorzuziehen, da sie die Cursorposition nicht ver.ndert.

Beispiele

```
WAIT "Bitte Taste dr.cken..." TO Taste
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

ZAP F.hrt die physische L.schung aller Datens.tze einer Datenbank durch

---

Syntax

ZAP

Anwendung

ZAP ist ein Datenbankbefehl, der den Inhalt der ge.ffneten Dateien im aktuellen Arbeitsbereich komplett und unwiederbringlich l.scht. Zu diesen Dateien z.hlen neben der selektierten Datenbank auch die offenen Indexdateien und eine evtl. vorhandene Memo-Datei. Der Platz auf der Platte, den die mit ZAP geleerten Dateien belegt haben, wird f.r DOS wieder freigegeben. ZAP hat dasselbe Resultat wie DELETE ALL gefolgt von PACK, ZAP ist jedoch wesentlich schneller.

Der Einsatz von ZAP in einer Netzwerkumgebung kann nur erfolgen, wenn die betroffenen Dateien mit USE EXCLUSIVE ge.ffnet worden sind. Weitere Informationen zur Programmierung von Netzwerkanwendungen finden Sie im Handbuch Programmierung und Hilfsprogramme.

Beispiele

Das folgende Beispiel zeigt eine typische ZAP-Operation in einer Netzwerkumgebung:

```
USE Verkauf EXCLUSIVE NEW
IF .NOT. NETERR()
  SET INDEX TO Verkauf, Filiale
  ZAP
  CLOSE Verkauf
ELSE
  ? "Datei kann nicht geleert werden!"
  BREAK
ENDIF
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

BEGIN SEQUENCE
Definiert eine Anweisungsfolge, die durch ein BREAK unterbrochen werden kann

---

Syntax

```
BEGIN SEQUENCE
  <Anweisungen>...
  [BREAK [<objError>]]
  <Anweisungen>...
[RECOVER [USING <idVar>]]
  <Anweisungen>...
END [SEQUENCE]
```

Parameter

BREAK <objError> verzweigt die Ausf.hrung zur n.chsten RECOVER-Anweisung (wenn vorhanden) oder zur n.chsten END SEQUENCE-Anweisung. Dabei ist <objError> ein Fehler-Objekt, das der folgenden RECOVER-Anweisung als Parameter .bergeben wird.

RECOVER USING<idVar> definiert einen Bezugspunkt, auf den ein Programm innerhalb eines BEGIN SEQUENCE...END-Konstruktes verzweigt, sobald eine BREAK-Anweisung erfolgte. Ist dabei eine Variable <idVar> angegeben, nimmt diese das von einer BREAK-Anweisung .bergebene Objekt als Parameter entgegen. Normalerweise ist dies ein Fehler-Objekt.

END SEQUENCE definiert das Ende eines BEGIN SEQUENCE-Konstruktes.

Anwendung

BEGIN SEQUENCE...END ist eine Kontrollstruktur, die zur Behandlung von Ausnahmen- und Fehlern verwendet werden kann. Sie begrenzt einen Anweisungsblock, einschlie.lich der Aufrufe von Prozeduren und anwenderdefinierten Funktionen (UDFs). Trifft die Programmfolge irgendwo innerhalb des durch BEGIN SEQUENCE und RECOVER begrenzten Anweisungsblockes auf ein BREAK, verzweigt das Programm zu der Zeile, die unmittelbar dem RECOVER-Statement folgt. Ist kein RECOVER-Statement angegeben, verzweigt das Programm in die Zeile, die unmittelbar auf das END-Statement folgt, um die Befehlsfolge innerhalb der Kontrollstruktur abzubrechen. Erreicht die Programmfolge ein RECOVER-Statement, ohne zuvor auf ein BREAK gesto.en zu sein, wird das Programm in der Zeile fortgef.hrt, die unmittelbar dem korrespondierenden END-Statement folgt.

Dem RECOVER-Statement kann optional ein Parameter .bergeben werden, der den Ergebniswert, eines in der BREAK-Anweisung angegebenen Ausdruckes beinhaltet. Meist ist dies ein Fehlerobjekt, das von der aktuellen Fehlerbehandlungsroutine erzeugt und zur.ckgegeben wird. An dieses Fehlerobjekt k.nnen Nachrichten zur Abfrage von Informationen .ber Fehlerursachen und .hnliche Informationen gesendet werden. Mit diesen Informationen k.nnen weitere Behandlungen eines aufgetretenen Laufzeitfehlers erfolgen (siehe Beispiel).

In einem BEGIN SEQUENCE-Konstrukt darf zwar eine RETURN-Anweisung zwischen dem BEGIN SEQUENCE- und RECOVER-Statement erfolgen, bei einigen anderen Statements bestehen jedoch Einschr.nkungen. So darf kein LOOP oder EXIT aus einer DO WHILE- oder FOR...NEXT-Schleife au.erhalb des BEGIN SEQUENCE-Blockes heraus erfolgen.

Nach einem RECOVER-Statement k.nnen EXIT-, LOOP- oder BREAK-Anweisungen angegeben werden. Ein LOOP aus einem RECOVER-Anweisungsblock ist sehr hilfreich, wenn eine SEQUENCE nochmals durchgef.hrt werden soll (Siehe Beispiel).

BEGIN SEQUENCE-Konstrukte sind sehr flexibel. Sie k.nnen verschachtelt werden und d.rfen in einer Prozedur oder einer anwenderdefinierten Funktion (UDF) mehrfach vorkommen. Werden mehrere SEQUENCE-Konstrukte

angegeben, begrenzt jedes Konstrukt einen getrennten Programmabschnitt.

Beispiele

. Dieser Programmabschnitt zeigt ein SEQUENCE-Konstrukt mit einem BREAK innerhalb der aktuellen Prozedur:

```
BEGIN SEQUENCE
  <Anweisungen>...
  IF lBreakCond
    BREAK
  ENDIF
  <Anweisungen>...
RECOVER
  <Wiederherstellungs-Anweisungen>...
END
```

. Das folgende Beispiel demonstriert, wie ein Fehler-Objekt von einer Fehlerbehandlungsroutine an eine Variable des RECOVER-Statements .bergeben wird:

```
LOCAL objLocal, bLastHandler
//
// Sichern des aktuellen und Definition eines neuen
// Fehlerbehandlungsblockes
bLastHandler := ERRORBLOCK( { |objErr | MyHandler(objErr, .T.) } )
//
BEGIN SEQUENCE
  .
  .<Anweisungen, die einen Fehler verursachen>...
  .
RECOVER USING objLocal
  //
  // Nachrichten an Fehler-Objekt senden und Fehler behandeln
  ? "Fehler: "

  IF objLocal:genCode <> 0
    ?? objLocal:description
  ENDIF
  .
  .
END
//
// Zur.cksetzen der vorherigen Fehlerbehandlung
ERRORBLOCK(bLastHandler)
FUNCTION MyHandler ( objError, lLocalHandler )
  //
  // Wenn Behandlung lokal, Fehlerobjekt zur.ckgeben
  IF lLocalHandler
    BREAK objError
  ENDIF
  .
  .<Andere Fehlerbehandlungen>...
  .
RETURN NIL
```

. Das n.chste Beispiel zeigt, wie ein SEQUENCE-Anweisungsblock mehrfach ausgef.hrt werden kann, wenn eine LOOP-Anweisung nach einem RECOVER-Statement erfolgt:

```
DO WHILE .T.
  BEGIN SEQUENCE
    .
    .<Anweisungen, die einen Fehler verursachen>...
    .
  RECOVER
    IF lPrintRecover
      LOOP  // Wiederholung der SEQUENCE-Anweisungen
    ENDIF
```

```
  END
  EXIT       // Abbruch der Operation
ENDDO
```

See Also:

DECLARE*

Anlegen und Initialisieren von Privat-Speichervariablen und -Arrays

---

Syntax

DECLARE <Bezeichner> [[:= <Initialisierung>], ...]

Parameter

<Bezeichner> ist der Name einer Private-Speichervariablen oder eines Arrays. Folgt dem Variablennamen <Bezeichner> ein eckiges Klammernpaar, wird ein Array erzeugt. Ist dies der Fall, wird die Anzahl der Elemente f.r jede Dimension wie folgt angegeben:

```
array[<nElemente>,<nElemente2>,...] oder
array[<nElemente>][<nElemente2>]...
```

Die maximale Anzahl der Elemente je Dimension betr.gt 4096.
<Initialisierung> ist eine optionale Wertzuweisung der neuen Variablen. Solch eine Zuweisung erfolgt durch den Zuweisungs-Operator (:=), gefolgt von einem beliebigen, aber g.ltigen Clipper-Ausdruck (einschlie.lich literaler Arrays). Erfolgt keine Wertzuweisung, wird die Variable mit dem Wert NIL initialisiert. Bei Arrays werden alle Elemente mit NIL initialisiert.

Mehrere Variablen oder Arrays k.nnen innerhalb einer DECLARE-Anweisung gleichzeitig angelegt und initialisiert werden, wenn sie durch Kommata getrennt werden.

Anwendung

DECLARE ist ein Kompatibilit.ts-Statement und ein Synonym f.r das PRIVATE-Statement. Aus diesem Grund wird die Verwendung dieses Statements nicht empfohlen. Das PRIVATE-Statement kann in allen F.llen ersatzweise angewendet werden. Weitergehende Informationen erhalten Sie in der Beschreibung des PRIVATE-Statements in diesem Kapitel.

See Also:

DO*
Aufruf einer Prozedur

Synax
DO <idProzedur> [WITH <Parameter Liste>]

Parameter
<idProzedur> ist der Name der auszuf.hrenden Prozedur.

WITH spezifiziert eine Reihe von Parametern <Parameter Liste>, die einzeln durch Kommata getrennt, der Prozedur .bergeben werden. Der aufzurufenden Prozedur k.nnen dabei bis zu 128 Parameter .bergeben werden. Jeder Parameter kann eine einzelne Variable, ein Feld, ein Array, ein Arrayelement, ein g.ltiger Clipper-Ausdruck oder ein Objekt sein. Einzelne Parameter innerhalb <Parameter Liste> k.nnen weggelassen werden.

Anwendung
Die DO-Anweisung ruft eine Prozedur auf, wobei der aufgerufenen Routine optionale Parameter .bergeben werden k.nnen. Die Durchf.hrung der Aktion ist identisch mit dem Aufruf einer anwenderdefinierten Funktion oder Prozedur, mit der Ausnahme, da. alle .ber- gebenen Variablen, bis auf Datenfelder, per Referenz .bergeben werden. In der Reihe der .bergebenen Parameter m.ssen Datenfelder in Klammern eingeschlossen sein, sofern sie nicht mit FIELD deklariert oder mit dem Aliasbezeichner angegeben werden. Unter Clipper kann die Anzahl der .bergebenen Argumente von der Anzahl der .ber das PARAMETERS-Statement entgegengenommenen Parameter abweichen. Ist die Anzahl der .bergebenen Argumente kleiner als die Anzahl der entgegengenommenen Parameter, werden bei Aufruf der Prozedur solche Parameter, die nicht mit einem .bergebenen Argument korrespondieren, mit NIL vorbelegt. .hnlich ist die Vorgehensweise bei Argumenten, die .bersprungen bzw. weggelassen wurden. Um festzustellen, wie viele Parameter .bergeben wurden, kann die Funktion CPCOUNT() verwendet werden. .bersprungene Parameter k.nnen durch einen Vergleich mit NIL identifiziert werden.

Des weiteren wirkt sich der Aufruf einer Prozedur oder anwenderdefinierten Funktion (UDF) .ber DO auch beim Compilieren eines Programmes aus. Wird beim Compilieren auf die Compilereinstellung /M verzichtet und ist eine gleichnamige Prozedur nicht in der aktuellen Quelldatei enthalten, sucht Clipper bei Auftreten eines DO-Statements eine Quelldatei (.prg) mit dem gleichen Namen, wie nach dem DO angegeben, und compiliert diese. Ist keine Datei mit gleichem Namen vorhanden, nimmt Clipper an, da. es sich um eine externe Prozedur handelt, und es wird ein entsprechender Verweis in die OBJ-Datei gestellt. Beim Linken der Dateien wird dieser Verweis dann in allen OBJ- und LIB-Dateien gesucht.

Unter Clipper 5.0 ist das DO-Statement nur aus Kompatibilit.tsgr.nden implementiert worden, der Gebrauch wird daher nicht empfohlen. Die Form des Aufrufes wird durch die direkte Angabe des Funktions- oder Prozedurnamens innerhalb einer Zeile ersetzt. Die Parameter werden dabei jedoch als Wert (per Value) .bergeben. Um Variablen trotzdem per Referenz zu .bergeben, mu. dem Variablennamen ein Klammeraffe (@) vorangestellt werden.

Damit Sie trotzdem ein DO-Statement in Ihren Programmen verwenden k.nnen, um zum Beispiel die Lesbarkeit der Programme zu erh.hen, ist innerhalb STD.CH ein entsprechendes anwenderdefiniertes Kommando .ber eine #command-Direktive implementiert, die den Aufruf entsprechend umsetzt.

Um mehr .ber die Parameter.bergabe zu erfahren, lesen Sie bitte den Abschnitt .ber Funktionen und Prozeduren im Kapitel Grundkonzepte.

Beispiele

. Dieses Beispiel f.hrt eine Prozedur ohne Parameter.bergabe aus:

```
DO Beispiel

BEISPIEL()              // Empfohlene Methode
```

. Dieses Beispiel f.hrt eine Prozedur mit zwei .bergebenen Konstanten aus:

```
DO Beispiel WITH "Demo", "Parameter.bergabe"

BEISPIEL("Demo", "Parameter.bergabe") // Empfohlene Methode
```

. In diesem Beispiel wird eine Prozedur ausgef.hrt, wobei der erste Parameter per Value und der zweite Parameter per Referenz .bergeben wird:

```
nNummer := 12
DO AlleMonate WITH nNummer + 12, nNummer
AlleMonate(nNummer + 12, @nNummer)  // Empfohlene Methode
```

. Hier wird eine Prozedur aufgerufen, wobei einzelne Parameter in der Parameterliste weggelassen wurden:

```
DO Berechnung WITH , , , Zahl1, Zahl2
Berechnung(, , , Zahl1, Zahl2)  // Empfohlene Methode
```

See Also:

DO CASE
F.hrt einen aus mehreren Anweisungsbl.cken aus

Syntax

```
DO CASE
  CASE <lBedingung1>
    <Anweisungen>...
  [CASE <lBedingung2>]
    <Anweisungen>...
  [OTHERWISE]
    <Anweisungen>...
END[CASE]
```

Parameter

CASE <lBedingung> definiert einen Anweisungsblock, der ausgef.hrt wird, wenn die Bedigung <lBedingung> den Wert true (.T.) ergibt.

OTHERWISE definiert einen Anweisungsblock, der ausgef.hrt wird, wenn keine der vorangegangenen CASE-Bedingungen true (.T.) ergeben hat.

Anwendung

DO CASE...ENDCASE ist eine Kontrollstruktur, die einen von mehreren Anweisungs-Bl.cken ausf.hrt, wenn die jeweilige Bedingung zutrifft. Dies erfolgt durch eine Programmverzweigung zu dem ersten CASE-Statement, dessen Bedingung true (.T.) ergibt. Die Anweisungen werden dann so lange ausgef.hrt, bis das n.chste CASE-Statement, OTHERWISE oder ENDCASE erreicht wird. Das Programm wird immer bei der Zeile fortgef.hrt, die dem n.chsten ENDCASE folgt.

Eine Reihe von Anweisungen, einschlie.lich anderer Kontrollstrukturen (z.B. DO WHILE, NEXT etc.), k.nnen in eine einzelne DO CASE-Struktur eingebunden und auch mit dieser verschachtelt werden. Es bestehen keine Limitierungen, was die Anzahl der CASE-Statements innerhalb einer DO CASE-Struktur betrifft.

Die Kontrollstruktur DO CASE...ENDCASE ist identisch mit IF...ELSEIF...ENDIF, wenn man von der unterschiedlichen Syntax absieht.

Beispiele

Das folgende Beispiel demonstriert ein Men., bei dem die von der Benutzereingabe abh.ngigen Programmverzweigungen durch ein DO CASE-Konstrukt vorgenommen werden:

```
  @ 3, 25 PROMPT "Erste Auswahl"
  @ 5, 25 PROMPT "Zweite Auswahl"
  MENU TO nWahl

  DO CASE
  CASE nWahl = 1
    Programm1()
  CASE nWahl = 2
    Programm2()
  OTHERWISE
    ? "Keine g.ltige Auswahl"
  ENDCASE
RETURN
```

See Also:

DO WHILE
F.hrt eine Schleife so lange aus, wie die Bedingung true (.T.) liefert

---

Syntax

```
[DO] WHILE <lBedingung>
  <Anweisungen>...
  [EXIT]
  <Anweisungen>...
  [LOOP]
  <Anweisungen>...
END[DO]
```

Parameter

<lBedingung> ist ein logischer Ausdruck, der als Bedingung f.r das Schleifenkonstrukt DO WHILE...ENDDO fungiert.

EXIT ist eine nicht bedingte Programmverzweigung innerhalb einer DO WHILE-Schleife hinter das n.chste, korrespondierende ENDDO-Statement.

LOOP ist eine unkonditionale Programmverzweigung innerhalb einer DO WHILE-Schleife auf den Anfang des Schleifenkonstruktes DO WHILE.

Anwendung

DO WHILE...ENDDO ist eine Kontrollstruktur, welche alle innerhalb dieser Struktur befindlichen Anweisungen so lange wiederholt, wie die zugeh.rige Bedingung erf.llt ist (true liefert). Die innerhalb der Schleife befindlichen Anweisungen werden ausgef.hrt, bis ein EXIT, LOOP oder ENDDO auftritt.

Ein ENDDO- oder ein LOOP-Statement f.hrt zu einer Programmverzweigung unmittelbar auf den Schleifenanfang, also zu den Anweisungen unmittelbar nach DO WHILE, und wiederholt die Anweisungen nochmals, falls die Schleifenbedingung erf.llt ist. Ansonsten wird die Schleife verlassen. Ein EXIT-Statement innerhalb der Schleife verzweigt direkt hinter das Ende der Programmschleife, also hinter das ENDDO. EXIT wird benutzt, um eine DO WHILE-Struktur unabh.ngig von der Schleifenbedingung zu verlassen. Im Gegensatz dazu verwendet man LOOP, um die Anweisungen der Schleife zu wiederholen, jedoch nur dann, wenn eine vorherige .berpr.fung der Schleifenbedingung den Wert true (.T.) ergibt. Erreicht wird dies durch eine Programmverzweigung auf den Schleifenanfang.

DO WHILE-Strukturen k.nnen mit anderen Kontrollstrukturen beliebig tief verschachtelt werden. Die einzige Anforderung besteht darin, da. jede Kontrollstruktur in sich abgeschlossen ist.

Beispiele

. Das folgende Beispiel demonstriert eine typische Kontrollstruktur f.r eine einfache Auswertung:

```
LOCAL cVerkAlt, nSubTotal
USE Verkauf INDEX Verkaeufer NEW
DO WHILE .NOT. EOF()              // Schleifenanfang mit Bedingung
  cVerkAlt := Verkauf->Verkaeufer
  nSubTotal := 0
  DO WHILE cVerkAlt == Verkauf->Verkaeufer .AND. .NOT. EOF()
    ? Verkauf->Verkaeufer, Verkauf->Umsatz
    nSubTotal := nSubTotal + Verkauf->Umsatz
    SKIP
  ENDDO
  ? "Summe: ", nSubTotal, " f.r " + Verkauf->Verkaeufer
ENDDO
CLOSE Verkauf
```

. Der folgende Programmteil zeigt, wie LOOP eingesetzt werden kann, um die Schleife, abh.ngig von einer zus.tzlichen Bedingung, unmittelbar zu

wiederholen:

```
DO WHILE <lBedingung>
  <Anweisungen>...
  IF <lUnmittelbareBedingung>
    LOOP
  ENDIF
  <Anweisungen>...
ENDDO
```

. Dieses Beispiel zeigt, wie mit einer DO WHILE-Schleife ein REPEAT...UNTIL-Konstrukt nachgebildet werden kann:

```
LOCAL lMehr := .T.

DO WHILE lMehr
  <Anweisungen>...
  lMehr := <lBedingung>
ENDDO
```

. Das folgende Beispiel zeigt eine Schleifenstruktur, um sequentiell eine Datenbank zu durchlaufen:

```
USE Kunden NEW

DO WHILE .NOT. EOF()
  <Anweisungen>...
  SKIP
ENDDO
```

See Also:

EXTERNAL
Deklariert eine Reihe von Prozedur- oder Funktionsnamen f.r den Linker

---

Syntax
EXTERNAL <idProzedur Liste>

Parameter
<idProzedur Liste> ist eine Liste mit Namen von Prozeduren, anwenderdefinierten Funktionen (UDFs) oder Formatprozeduren (.fmt). Diese Symbole werden dann beim Linklauf als extern angesehen und somit in allen Objekt-Dateien und Bibliotheken gesucht.

Anwendung
EXTERNAL ist eine Deklarations-Anweisung, die eine Referenz f.r den Linker deklariert. Wie bei allen anderen Deklarations-Anweisungen mu. das EXTERNAL-Statement angegeben werden, bevor irgendeine andere, ausf.hrbare Anweisung in einem Programm, einer Prozedur oder einer anwenderdefinierten Funktion erfolgt.

W.hrend der Compilation eines Clipper-Quellprogrammes wird eine spezielle Symbolliste aller Prozeduren und Funktionen f.r den Linker angelegt. In einigen F.llen kann der Compiler solche Referenzen nicht finden, da sie z.B. erst zur Laufzeit des Progammes gebildet werden. Mit EXTERNAL werden solche Referenzen vorgegeben, indem die Namen der Prozeduren und Funktionen in die Referenzliste aufgenommen werden. So wird der Linker veranla.t, diese Symbole und die damit verbundenen Routinen mit einzubeziehen. Die EXTERNAL-Deklaration ist in folgenden F.llen wichtig:

. Beim Aufruf von Prozeduren, anwenderdefinierten Funktionen oder Formatprozeduren innerhalb von Makro-Ausdr.cken.

. Bei Prozeduren und anwenderdefinierten Funktionen, die in REPORT- und LABEL-Dateien benutzt werden und nicht auf bereits bestehende Routinen im Quellprogramm verweisen.

. Bei anwenderdefinierten Funktionen, die in Indexschl.sseln eingesetzt werden und im Quellprogramm nicht vorhanden sind.

. Bei Anwendungsfunktionen, die in ACHOICE(), DBEDIT() oder MEMOEDIT angegeben sind.

Um eine Gruppe von immer wieder ben.tigten EXTERNAL-Anweisungen einzubinden, k.nnen diese als Definitionen in eine Header-Datei aufgenommen werden, welche dann .ber #include an das jeweilige Quellprogramm gebunden wird.

Beispiele
Dieses Beispiel zeigt eine typische Header-Datei, in der EXTERNAL-Deklarationen vorgenommen werden:

```
// External.ch
EXTERNAL HARDCR, TONE
EXTERNAL MEMOTRAN, STRTRAN
```

See Also:

FIELD
Deklariert Namen von Datenbankfeldern

Syntax
FIELD <idFelder Liste> [IN <idAlias>]

Parameter
<idFelder Liste> entspricht einer Reihe von zu deklarierenden Feldnamensbezeichnern.

IN <idAlias> gibt optional einen Aliasbezeichner an, wenn auf Felder in nicht-selektierten Arbeitsbereichen verwiesen werden soll. Wenn diese Option nicht angegeben ist, werden die in <idFelder Liste> angegebenen Feldnamen so behandelt, als w.rde ein Feldalias (FIELD->) vor dem Namen stehen.

Anwendung
Das FIELD-Statement deklariert Namen von Datenbankfeldern f.r den Compiler und unterst.tzt dabei optional Feldnamen aus anderen Aliasbereichen. Dies erm.glicht dem Compiler, Verweise auch auf solche Variablen zu bilden, die ohne ausdr.ckliche Aliasbezeichner angegeben wurden, um sie so einem bestimmten Datenfeld zuzuordnen. Es werden nur Referenzen mit Namen aus <idFelder Liste> gebildet, die im weiteren Programm nicht ausdr.cklich mit einem Aliasbezeichner versehen sind. Das FIELD-Statement hat, wie auch andere Deklarationen, keine Auswirkung, wenn es in einem Makro-Ausdruck oder in einer Variablen enthalten ist.

Das FIELD-Statement .ffnet weder eine Datenbank, noch verifiziert es die Existenz der Datenfelder. Es wird ausschlie.lich ben.tigt, um korrekte Referenzen zu bilden. Damit ist garantiert, da. zur Laufzeit eines Programmes auf die richtigen Felder zugegriffen wird. Auf die Feldvariablen selbst kann zur Laufzeit nur zugegriffen werden, wenn die Datei zuvor mit einem USE ge.ffnet worden ist. Ein Zugriff auf Feldvariablen einer nicht ge.ffneten Datenbank ruft einen Laufzeitfehler hervor.

Der Geltungsbereich einer FIELD-Deklaration ist die Funktion oder Prozedur, innerhalb derer sie angegeben wurde. Sie kann auch f.r eine ganze .PRG-Datei gelten, wenn sie vor alle Funktions- und Prozedurdeklarationen gestellt und der Compiler mit der Option /N aufgerufen wird.

Ein FIELD-Statement mu., wie auch andere Deklarationen, in einer Prozedur, Funktion oder .PRG-Datei vor allen ausf.hrbaren Anweisungen stehen.

FIELD-Deklarationen werden im Zusammenhang mit dem Compilerschalter /W benutzt, um nach allen nicht deklarierten Variablen zu suchen. Weitere Informationen .ber die Variablen-Deklaration und

Geltungsbereiche finden Sie im Abschnitt Variablen des Kapitels Grundkonzepte.

Beispiele
Die folgende anwenderdefinierte Funktion (UDF) enth.lt Anweisungen, um Datenbankfelder aus zwei Arbeitsbereichen zu deklarieren:

```
FUNCTION DisplayRecord
  FIELD KundenNr, Auftr_Nr, Verkaeufer
  FIELD Name, Adresse, Telefon IN Kunden
  USE Kunden INDEX KundenNr NEW
  USE Aufrag NEW
  SET RELATION TO Kunde INTO Kunden
  //
  ? Auftr_Nr         // Verweist auf Auftrag->Auftr_Nr
  ? Name             // Verweist auf Kunden->Name
  //
```

```
  CLOSE Kunden
  CLOSE Auftrag
RETURN NIL
```

See Also:

ALIAS() Ermittelt den Aliasnamen eines Arbeitsbereiches

Syntax

ALIAS([<nArbeitsbereich>]) -> cAlias

Parameter

<nArbeitsbereich> ist die Nummer des Arbeitsbereiches, dessen Aliasname ermittelt werden soll.

R.ckgabe

ALIAS() erzeugt als R.ckgabewert eine Zeichenfolge, die den Aliasnamen des angegebenen Arbeitsbereiches enth.lt. Wird kein Arbeitsbereich in <nArbeitsbereich> angegeben, wird der Name des aktuellen Arbeitsbereiches zur.ckgegeben. Ist im Arbeitsbereich keine Datenbank ge.ffnet, wird eine leere Zeichenfolge "" zur.ckgegeben.

Anwendung

ALIAS() wird benutzt, um den Aliasnamen eines Arbeitsbereiches zu ermitteln. Der Aliasname ist die Bezeichnung, die ein Arbeitsbereich bekommt, sobald eine Datenbank ge.ffnet worden ist. Der tats.chliche Name eines Arbeitsbereiches ist entweder der Name der Datenbank selber, oder der mit dem Befehlsteil ALIAS angegebene Name.

ALIAS() ist die umgekehrte Funktion von SELECT(). W.hrend bei ALIAS() die Nummer des Arbeitsbereiches angegeben wird, um den Namen zu erfahren, wird bei SELECT() der Name angegeben, um die Nummer des Arbeitsbereiches zu ermitteln.

Beispiele

In diesem Beispiel werden zwei Datenbanken in verschiedenen Bereichen ge.ffnet und dann ALIAS() benutzt, um die Namen anzuzeigen:

```
USE Verkauf IN 1
USE Kunden  IN 2
? ALIAS(1), ALIAS()    // Resultat: "Verkauf Kunden"
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

FOR...NEXT

F.hrt einen Anweisungsblock in der angegebenen Anzahl von Wiederholungen aus

---

Syntax

```
FOR <idZ.hler> := <nStart> TO <nEnde> [STEP <nErh.hung>]
  <Anweisungen>...
  [EXIT]
  <Anweisungen>...
  [LOOP]
  <Anweisungen>...
NEXT
```

Parameter

<idZ.hler> ist der Name einer zur Schleifenkontrolle ben.tigten Z.hlervariable. Ist die Variable zu diesem Zeitpunkt nicht sichtbar oder extistiert noch keine Variable mit diesem Namen, wird sie als private Variable angelegt.

<nStart> ist der Anfangswert, den die Variable <idZ.hler> zugewiesen bekommt. Ist die Angabe in <nErh.hung> negativ, mu. <nStart> gr..er als <nEnde> sein.

TO <nEnde> definiert den Endwert der Variablen <idZ.hler>. Ist <nErh.hung> negativ, mu. <nStart> gr..er sein als <nEnde>, ansonsten umgekehrt.

STEP <nErh.hung> definiert den Wert, um den die Z.hlervariable bei jedem Schleifendurchlauf <idZ.hler> erh.ht wird. Dieser Wert kann positiv oder negativ sein. Ist die STEP-Option nicht angegeben, wird <idZ.hler> bei jedem Schleifendurchlauf um 1 erh.ht.

EXIT ist eine nichtbedingte Programmverzweigung innerhalb einer FOR...NEXT-Schleife hinter das n.chste, korrespondierende NEXT-Statement.

LOOP ist eine nichtbedingte Programmverzweigung innerhalb einer FOR...NEXT-Schleife auf den Anfang des Schleifenkonstruktes, also zur n.chsten Anweisung hinter dem FOR.

Anwendung

FOR...NEXT ist eine Kontrollstruktur, durch die alle innerhalb dieser Struktur befindlichen Anweisungen so oft wiederholt werden, wie Schritte definiert sind. Die Kontrollstruktur beginnt bei dem Anfangswert in <nStart> und erh.ht bei jedem Schleifendurchlauf den Wert in <idZ.hler> um <nErh.hung>, bis der Grenzwert in <nEnde> erreicht ist. F.r jeden Schleifendurchlauf werden alle Ausdr.cke im FOR-Statement neu berechnet. Die Ausdr.cke in <idZ.hler> und <nEnde> k.nnten daher innerhalb der Schleife durch beliebige Operationen ver.ndert werden.

Die Programmschleife FOR...NEXT wird so oft durchlaufen, bis entweder <idZ.hler> einen gr..eren Wert annimmt als <nEnde>, oder aber ein EXIT-Statement erreicht wird. Das Programm wird dann in der Programmzeile fortgef.hrt, die dem korrespondierenden NEXT-Statement folgt. Tritt ein LOOP-Statement auf, verzweigt das Programm zur.ck auf die Anweisung, die unmittelbar dem zugeh.rigen FOR-Statement folgt.
Ist in <nErh.hung> ein negativer Wert angegeben, wird die Z.hlervariable bei jedem Schleifendurchlauf um diesen Wert reduziert anstatt erh.ht. Die FOR...NEXT-Schleife wird dann so oft wiederholt, bis <idZ.her> einen niedrigeren Wert als <nEnde> annimmt. Dies bedeutet jedoch, da. der Startwert <nStart> gr..er sein mu. als der Wert in <nEnde>.
FOR...NEXT-Schleifen sind zum Beispiel bei der Abarbeitung von Arrays hilfreich, wobei <idZ.hler> als Index auf ein Arrayelement dienen kann.
FOR...NEXT-Strukturen k.nnen mit anderen Kontrollstrukturen beliebig tief verschachtelt werden. Die einzige Anforderung besteht darin, da. jede Kontrollstruktur in sich korrekt abgeschlossen wird.

Beispiele

. Dieses Beispiel durchl.uft ein Array in aufsteigender Folge:

```
nLenArray := LEN(aArray)

FOR i := 1 TO nLenArray
  ? aArray[i]
NEXT
```

. Das gleiche Beispiel, jedoch in absteigender Folge:

```
nLenArray := LEN(aArray)

FOR i := nLenArray TO 1 STEP -1
  ? aArray[i]
NEXT
```

See Also:

FUNCTION
Deklariert den Namen einer anwenderdefinierten Funktion und deren formale Parameter

---

Syntax

```
[STATIC] FUNCTION <idFunktion>[(<idParameter Liste>)]
  [LOCAL                          <Bezeichner> [[:= <Initialisierung>],... ]]
  [STATIC                                                                    u<B
  [FIELD                          <idFelder Liste> [IN <idAlias>]]
  [MEMVAR                                                                    u<i
  .
  . <ausf.hrbare Anweisungen>
  .
RETURN <exp>
```

Parameter

<idFunktion> ist der Name der zu deklarierenden anwenderdefinierten Funktion ((U)ser (D)efined (F)unction). Der Name der Funktion kann beliebig lang sein, es werden allerdings nur die ersten 10 Zeichen ber.cksichtigt. Namen k.nnen aus einer Kombination von alphabetischen Zeichen, Ziffern und Unterstrichen gebildet werden. Sie d.rfen jedoch nicht mit einem Unterstrich beginnen, da solche Namen unter Clipper reserviert sind.

<idParameter Liste> ist die Deklaration einer oder mehrerer Parameter-Variablen. Parameter-Variablen in dieser Liste sind automatisch als LOCAL spezifiziert.

[STATIC] FUNCTION deklariert eine anwenderdefinierte Funktion (UDF) als statisch, so da. sie nur von Prozeduren und Funktionen innerhalb derselben Programmdatei (.PRG) aufgerufen werden kann.

LOCAL deklariert eine Reihe von Variablen oder Arrays als LOCAL, wobei optional in der gleichen Anweisung Werte zugewiesen werden k.nnen. Existenz und Sichtbarkeit sind auf die deklarierende Funktion beschr.nkt.
STATIC deklariert eine Reihe von Variablen oder Arrays als STATIC, wobei optional in der gleichen Anweisung Werte zugewiesen werden k.nnen. Die Sichtbarkeit ist auf die deklarierende Funktion eingeschr.nkt, w.hrend sich die Existenz auf die Dauer des gesamten Programmes bezieht.

FIELD deklariert eine Reihe von Bezeichnern als Feldnamen, wobei IN <Alias> optional den Arbeitsbereich angibt, f.r den diese Feldbezeicher gelten sollen.

MEMVAR deklariert eine Reihe von Bezeichnern als PRIVATE- oder PUBLIC-Speichervariablen.

RETURN <exp> verl..t die aktuelle Funktion und gibt den Wert von <exp> an die aufrufende Funktion zur.ck. Jede Funktion mu. mindestens ein RETURN-Statement enthalten und einen R.ckgabewert besitzen. Ein RETURN-Statement kann innerhalb des Anweisungsblockes der Funktion an einer beliebigen Stelle auftreten.
Beachten Sie bitte, da. dies nicht unbedingt den Regeln der strukturierten Programmierung entspricht.

Anwendung

Das FUNCTION-Statement deklariert eine anwenderdefinierte Funktion (User Defined Function oder abgek.rzt auch UDF) und optional eine Reihe von lokalen Variablen, die .bergebene Parameter als formale Parameter entgegennehmen. Eine anwenderdefinierte Funktion ist ein Unterprogramm mit darin enthaltenen Deklarationen und Anweisungen, die ausgef.hrt werden, wenn ein Name <idFunktion>, gefolgt von einem Klammernpaar, im Programm zur Ausf.hrung kommt.

Eine Funktionsdefinition beginnt mit dem FUNCTION-Statement und endet

mit dem n.chsten FUNCTION- oder PROCEDURE-Statement oder aber mit dem Ende der Programmdatei.

Funktionen dienen dazu, immer wiederkehrende Teile von Programmcode zusammenzufassen, um darauf in beliebigen Ausdr.cken durch Verwendung des Funktionsnamens zuzugreifen. Funktionen und Prozeduren erh.hen die Lesbarkeit, Modularit.t, Programmpflege und Wartbarkeit gr..erer Programme und helfen, diese einfacher zu organisieren.

Unter Clipper sind Funktionen das gleiche wie Prozeduren, mit der Ausnahme, da. Funktionen einen R.ckgabewert liefern m.ssen. Der zur.ckgegebene Wert kann beliebigen Typs sein, einschlie.lich Arrays, Code-Bl.cken oder NIL. Jede Funktion mu. mit einem FUNCTION-Statement beginnen und mindestens ein RETURN mit einem Argument als R.ckgabewert enthalten. Funktions-Deklarationen d.rfen nicht mit anderen Funktions-Deklarationen verschachtelt werden. Eine anwenderdefinierte Funktion kann wie jede andere Funktion unter Clipper angewendet werden. Die Sichtbarkeit von Funktionsnamen zerf.llt in zwei Klassen.

Funktionen, die im gesamten Programm zur Verf.gung stehen sollen, bezeichnet man als Public-Funktionen; sie werden einfach .ber das FUNCTION-Statement deklariert. Funktionen, die nur innerhalb eines Programmteiles (.PRG-Datei) sichtbar sein sollen, werden als Static-Funktionen bezeichnet und .ber STATIC FUNCTION deklariert. Static Function sagt aus, da. sie einen Datei- weiten G.ltigkeitsbereich (file-wide scope) besitzen.

STATIC Funktions-Deklarationen sind in verschiedenen Situationen sehr hilfreich. Zum ersten begrenzen sie die Sichtbarkeit und daher auch den Zugriff auf solche Funktionen. Aus diesem Grund k.nnen Teilprogramme innerhalb einer Programmdatei allgemein zugreifbare Funktionen (public functions) bereithalten, ohne jedoch Implementierungsdetails, definiert als STATIC-Funktionen oder -Prozeduren, preiszugeben. Des weiteren werden Referenzen auf static Functions bereits zur Compilierzeit aufgel.st. Dies vermeidet Konflikte mit Funktionsnamen erst zur Linkzeit aufgel.st werden. Damit wird sichergestellt, da. statische Funktionen innerhalb der gleichen Programmdatei auch dann aufgerufen werden k.nnen, wenn in anderen Programmdateien Public-Funktionen mit gleichen Namen vorkommen. Weitere Informationen .ber anwenderdefinierte Funktionen (UDF), Variablen-Deklaration und die Parameter.bergabe finden Sie im Kapitel Grundkonzepte.

Hinweise

. Aufruf einer anwenderdefinierten Funktion: Eine anwenderdefinierte Funktion wird in der gleichen Weise wie eine Clipper-Funktion aufgerufen:

<idFunktion>([<Argumentenliste>])

Eine anwenderdefinierte Funktion kann innerhalb eines Ausdruckes oder allein in einer Zeile aufgerufen werden. In letzterem Fall wird der R.ckgabewert ignoriert.
Eine anwenderdefinierte Funktion kann auch mit einem Aliasbezeichner aufgerufen werden. Der Aliasbezeichner wird dabei vorangestellt und der Funktionsausdruck in Klammern eingegrenzt:

<idAlias>->(<idFunktion>([<Argumentenliste>]))

Wird eine Funktion in einem Alias-Ausdruck aufgerufen, gilt f.r diesen Ausdruck der Arbeitsbereich, auf den der Aliasbezeichner verweist. Es wird also f.r die Dauer des Ausdruckes tempor.r der angegebene Arbeitsbereich selektiert. .hnlich wie bei anderen Ausdr.cken kann ein Alias-Ausdruck separat in einer Zeile stehen.
Eine anwenderdefinierte Funktion kann sich unter Clipper selber aufrufen (rekursiver Aufruf). Dies bedeutet, da. innerhalb der ausf.hrbaren Anweisungen einer Funktion ein Verweis auf dieselbe Funktion erfolgen kann. Es ist jedoch Vorsicht geboten, da ein Laufzeitfehler erfolgen wird (Stack-.berlauf), wenn keine geeignete Abbruchbedingung vorgesehen ist.

. Parameter: Anwenderdefinierte Funktionen sowie Prozeduren k.nnen vom aufrufenden Proze. (Prozedur, Funktion oder DOS-Kommandointerpreter) Parameter entgegennehmen. Ein Parameter ist ein Platzhalter f.r einen Wert oder eine Referenz. Unter Clipper 5.0 bestehen zwei M.glichkeiten, Parameter entgegenzunehmen: als Liste lokaler Variablennamen in der Deklarationszeile der Funktion oder als Liste von Privat-Variablennamen in einer PARAMETERS-Anweisung. Beachten Sie dabei bitte, da. Sie diese beiden Methoden nicht mischen d.rfen, dies w.rde einen schwerwiegenden Compilier-Fehler hervorrufen.
Parameter werden in der gleichen Reihenfolge entgegengenommen, wie sie auch .bergeben worden sind. Unter Clipper darf die Anzahl der entgegengenommenen Parameter von der Anzahl der .bergebenen Argumente abweichen. Einzelne der zu .bergebenden Argumente k.nnen innerhalb der Argumentenliste weggelassen werden. Ein Parameter, der nicht .bergeben wurde, ist mit NIL vorinitialisiert. Die Anzahl der .bergebenen Argumente kann .ber PCOUNT() bestimmt werden, wobei .bersprungene Argumente mitgez.hlt werden.
Parameter k.nnen als Referenz oder als Wert an eine anwenderdefinierte Funktion .bergeben werden. Die Standardmethode f.r Ausdr.cke und Variablen ist eine .bergabe als Wert (per value). Dies schlie.t Variablen, die eine Referenz auf Arrays oder Code-Bl.cke enthalten, ein. Variablen, mit Ausnahme von Datenfeldern, k.nnen durch Voranstellen des @-Operators per Referenz .bergeben werden.

Beispiele

. Dieses Beispiel veranschaulicht eine anwenderdefinerte Funktion zur Formatierung von numerischen Werten einschlie.lich W.hrungssymbol:

```
#define International = .T.

? DisplValuta(1000) // Resultat: $1,000.00

FUNCTION DisplValuta( nWert )
  LOCAL cZahl
  IF International
     cZahl := TRANSFORM(nWert,"@( $99,999,999.99")
  ELSE
     cZahl := TRANSFORM(nWert,"@E( 999,999,999.99DM")
  ENDIF
RETURN cZahl
```

. Das n.chste Beispiel zeigt eine anwenderdefinierte Funktion, die eine Zeichenkette mit durch Kommata getrennten Wertangaben entgegennimmt, und konvertiert diese in ein Array:

```
cString := "Eins,Zwei,Drei"
aListe  := ListeAlsArray(cString) // Resultat: {"Eins","Zwei","Drei"}

FUNCTION ListeAlsArray( cListe )
  LOCAL nPos
  LOCAL aListe := {}  // Anlegen eines leeren Arrays
  //
  DO WHILE (nPos := AT(",", cListe)) <> 0
     AADD(aListe, SUBSTR(cListe, 1, nPos -1))
     cListe := SUBSTR(cListe, 1, nPos +1)
  ENDDO
  AADD(aListe, cListe)
  //
RETURN aListe
```

. Das folgende Beispiel demonstriert, wie festgestellt werden kann, ob einzelne Argumente beim Aufruf weggelassen wurden:

```
FUNCTION Test( Param1, Param2, Param3 )
  IF Param2 == NIL     // Test des zweiten Parameters
     Param2 := "Vorgabewert"
  ENDIF
```

```
__.
  . <Anweisungen>...
__.
RETURN NIL
```

. Dieses Beispiel zeigt, wie die anwenderdefinierte Funktion DisplValuta() mit einem Aliasbezeichner angewendet werden kann:

```
USE Rechnung NEW
USE Kunden NEW
//
? Rechnung->(DisplValuta(EndBetrag))
```

See Also:

IF

F.hrt einen von verschiedenen Anweisungsbl.cken aus

Syntax

```
IF <lBedingung1>
  <Anweisungen>...
[ELSEIF <lBedingung2>]
  <Anweisungen>...
[ELSE]
  <Anweisungen>...
END[IF]
```

Parameter

<lBedingung> ist ein logischer Kontrollausdruck. Ergibt er true (.T.), werden die folgenden Anweisungen ausgef.hrt, bis ein ELSEIF, ELSE oder ENDIF erreicht wird.

ELSEIF definiert einen Block von Anweisungen, der ausgef.hrt wird, wenn die Bedingung in diesem Statement <lBedingung> den Wert true (.T.) ergibt und vorangegangene Bedingungen in IF oder ELSEIF false (.T.) lieferten. Eine IF...ENDIF-Kontrollstruktur kann beliebig viele ELSEIF-Statements enthalten.

ELSE definiert einen Block von Anweisungen, der ausgef.hrt wird, wenn keine der vorangegangenen IF- oder ELSEIF-Bedingungen true (.T.) ergeben hat.

Anwendung

IF...ENDIF ist eine Kontrollstruktur, die einen von verschiedenen Anweisungsbl.cken ausf.hrt, wenn die entsprechende Bedingung erf.llt ist. Dies erfolgt durch eine Programmverzweigung zu dem ersten IF- oder ELSEIF-Statement, dessen Bedingung true (.T.) ergibt. Die Anweisungen werden dann so lange ausgef.hrt, bis das n.chste ELSEIF-, ELSE- oder ENDIF-Statement erreicht wird. Das Programm wird in der Zeile fortgef.hrt, die dem n.chsten ENDIF folgt.
Ergibt keine der IF- oder ELSEIF-Bedingungen true (.T.), werden die Anweisungen des ELSE-Blockes ausgef.hrt. Ist kein ELSE-Statement angegeben, wird das Programm in der Zeile fortgef.hrt, die dem ENDIF-Statement folgt.

IF...ENDIF-Strukturen k.nnen mit weiteren IF...ENDIF-Strukturen oder anderen Kontrollstrukturen verschachtelt werden. Sie m.ssen nur jeweils in sich abgeschlossen sein.

Die Kontrollstruktur IF...ELSEIF...ENDIF ist identisch mit DO CASE...ENDCASE, wenn man von der unterschiedlichen Syntax absieht.

Beispiele

Das folgende Beispiel pr.ft eine Reihe von Bedingungen innerhalb einer IF...ELSEIF..
ENDIF-Struktur:

```
LOCAL nNummer := 0
//
IF nNummer < 50
  ? "Kleiner 50"
ELSEIF nNummer = 50
  ? "Gleich 50"
ELSE
  ? "Gr..er 50"
ENDIF
```

See Also:

LOCAL
Deklariert und initalisiert lokale Variablen und Arrays

Syntax
LOCAL <Bezeichner> [[:= <Initialisierung>], ...]

Parameter
<Bezeichner> ist der Name einer Variablen oder eines Arrays, die bzw. das als LOCAL deklariert werden soll. Folgt dem Namen <Bezeichner> ein eckiges Klammernpaar ([]), wird ein Array deklariert. Die Syntax zur Angabe mehrdimensionaler Arrays kann wie folgt aussehen:

aArray[<nElemente1>,<nElemente2>,...] oder
aArray[<nElemente1>][<nElemente2>]...

Die maximale Anzahl von Elementen je Dimension betr.gt 4096. Die Anzahl der Dimensionen wird nur durch den verf.gbaren Speicher begrenzt.

<Initialisierung> ist eine optionale Wertzuweisung f.r eine lokale Variable. Arraybezeichnern k.nnen keine Werte .ber <Initialisierung> zugewiesen werden. Die <Initialisierung> einer Variablen mu. durch einen Zuweisungsoperator (:=), gefolgt von einem g.ltigen Clipper-Ausdruck oder einem literalen Array erfolgen. Wird keine <Initialisierung> angegeben, so wird die Variable mit NIL vorinitialisiert. Im Falle eines Arrays erh.lt jedes Element den Wert NIL.

Hinweis
Der Makro-Operator (&) darf in einer LOCAL-Deklaration nicht verwendet werden.

Anwendung
LOCAL ist eine Deklarations-Anweisung, die eine oder mehrere Variablen oder Arrays zur lokalen Verwendung innerhalb der aktuellen, anwenderdefinierten Funktion oder Prozedur bestimmt. Die Deklaration mu. vor einer PRIVAT-, PUBLIC-, PARAMETERS- oder einer anderen ausf.hrbaren Deklaration erfolgen. Eine lokale Variable versteckt alle ansprechbaren Private- oder sichtbaren Public-Variablen, wenn sie den gleichen Bezeichner (Variablennamen) tragen. Werden in einem LOCAL-Statement Variablen deklariert, die bereits deklariert sind, kommt es zu einem schwerwiegenden Compilier-Fehler (es wird keine .OBJ-Datei erzeugt). Diese Fehler k.nnen auftreten, wenn zwei Deklarationen mit identischem Variablennamen in der gleichen Routine oder als Re-Deklaration in einer Datei-.bergreifenden Deklaration erfolgen. Dies betrifft die Deklarationen durch FIELD, MEMVAR und STATIC.

Lokale Variablen sind nur in der Prozedur oder anwenderdefinierten Funktion sichtbar, in der sie deklariert wurden. Im Gegensatz zu Private-Variablen sind sie in von dort aus aufgerufenen Funktionen oder Prozeduren nicht sichtbar. Lokal deklarierte Variablen werden automatisch erzeugt, wenn die deklarierende Prozedur aufgerufen wird. Die Existenz und der Wert einer Variable beginnt beim Aufruf der Prozedur oder anwenderdefinierten Funktion und endet, wenn die Kontrolle durch RETURN an die aufrufende Ebene zur.ckgegeben wird. Erfolgt ein rekursiver Aufruf (die Funktion ruft sich selber auf), werden f.r jeden Aufruf neue Local-Variablen erzeugt.

Die Vorinitialisierung von lokalen Variablen oder Elementen lokaler Arrays ist der Wert NIL, wenn keine andere explizite Initialisierung angegeben ist. Die Initialisierung kann .ber jeden g.ltigen Clipper-Ausdruck einschlie.lich Funktionsaufrufen erfolgen.
Die maximale Anzahl lokaler Variablen ist in einem Programm nur durch den zur Verf.gung stehenden Speicher begrenzt. Als lokal deklarierte Arrays d.rfen je Dimension maximal 4096 Elemente enthalten.

Weitere Informationen .ber die Variablendeklaration und .ber deren

G.ltigkeitsbereiche entnehmen Sie bitte dem Abschnitt .ber Variablen im Kapitel Grundkonzepte.

Hinweise

. .berpr.fung lokaler Variablen innerhalb des Debuggers: Sollen lokale Variablen innerhalb des Clipper-Debuggers anprechbar sein, m.ssen die Programmdateien (.PRG) mit dem Compilerschalter /B compiliert werden, damit die zus.tzlich ben.tigten Informationen in die Objekt-Datei (.OBJ) aufgenommen werden.

. Lokale Parameter: Eine Reihe lokaler Parameter k.nnen innerhalb der FUNCTION- oder PROCEDURE-Anweisungszeile als Parameterliste angegeben werden:

```
FUNCTION <idFunktion>(<idParameter Liste>)
```

Die Deklaration lokaler Parameter hat Vorrang vor der Deklaration von private-Parametern durch das PARAMETERS-Statement.

. Makcro-Ausdr.cke: Lokale Variablen k.nnen nicht innerhalb von Makro-Variablen oder Ausdr.cken angegeben werden. Existieren Private- oder Public-Variablen mit gleichen Namen wie lokale Variablen, werden diese in Makro-Ausdr.cken verwendet. Existieren solche Variablen nicht, wird ein Laufzeitfehler erzeugt.

. MEMORY-Dateien: Lokale Variablen k.nnen nicht in Memory-Dateien (.MEM) gespeichert oder aus solchen entnommen werden.

. Typ lokaler Variablen: Die Funktion TYPE() verwendet einen Makro-Operator (&), um den Typ des .bergebenen Argumentes zu ermitteln. Diese Funktion kann daher nicht zur Ermittlung des Typs von lokalen oder statischen Variablen herangezogen werden. Verwenden Sie stattdessen die Funktion VALTYPE().

Beispiele

. Das folgende Beispiel deklariert zwei lokale Arrays und zwei lokale Variablen:

```
LOCAL aArray1[20, 10], aArray2[20][10], var1, var2
```

. Dieses Beispiel deklariert und initialisiert zwei lokale Variablen. Die erste Variable wird mit einem Datum initialisiert, die zweite mit einem literalen Array:

```
LOCAL dAktuell := DATE()
LOCAL aObst    := { ".pfel", "Birnen", "Kirschen" }
```

See Also:

MEMVAR
Deklariert Namen von Private- und Public-Variablen

Syntax
MEMVAR <idMemvar Liste>

Parameter
<idMemvar Liste> ist eine Reihe zu deklarierender Private- und Public-Variablennamen.

Anwendung
MEMVAR ist ein Deklarations-Statement. Es teilt dem Compiler mit, da. sich Variablen-Referenzen, die nicht explizit mit einem Aliasbezeichner versehen sind, auf Speichervariablen beziehen. Dies hat den gleichen Effekt, als w.rde den Variablen ein MEMVAR-Aliasbezeichner vorangestellt. Es werden also nur Referenzen auf die Variablen ber.cksichtigt, welche nicht explizit mit einem Aliasbezeichner versehen sind. Wie bei anderen Deklarations-Statements hat auch MEMVAR keinen Einflu. auf Referenzen, die in Makro-Ausdr.cken angegebenen wurden.

Das MEMVAR-Statement erzeugt weder Variablen, noch verifiziert es deren Existenz. Es wird prim.r ben.tigt, um korrekte Referenzen auf Variablen zu bilden. Zur Laufzeit m.ssen diese Variablen dann .ber PRIVATE, PARAMETERS oder PUBLIC angelegt werden. Dies kann in der Prozedur geschehen, die das MEMVAR-Statement enth.lt, aber auch in allen h.heren Ebenen. Jeder Zugriff auf eine Variable, die nicht zuvor angelegt wurde, ruft einen Laufzeitfehler hervor.

Der Geltungsbereich einer MEMVAR-Deklaration ist die Funktion oder Prozedur, in der sie angegeben wurde. Sie gilt dann f.r eine ganze .PRG-Datei, wenn sie den Funktions- und Prozedurdeklarationen vorangestellt und der Compiler mit der Option /N aufgerufen wird. Die /N-Option unterdr.ckt die automatische Definition einer Prozedur mit gleichen Namen wie die Programm-Datei (.PRG).

Ein MEMVAR-Statement mu., wie auch andere Deklarationen, in einer Prozedur, Funktion oder .PRG-Datei vor allen ausf.hrbaren Anweisungen, einschlie.lich PARAMETERS, PUBLICS und PRIVATE, erfolgen.

MEMVAR-Deklarationen k.nnen in Verbindung mit dem Compilerschalter /W benutzt werden, um nach undeklarierten Variablen zu suchen und entsprechende Warnungen auszugeben. Dies kann hilfreich sein, um in einem Programm nachtr.glich alle Variablen zu deklarieren.

Weitere Informationen .ber die Variablen-Deklaration und deren Geltungsbereich enth.lt der Abschnitt .ber Variablen im Kapitel Grundkonzepte innerhalb dieses Buches.

Beispiele
Das folgende Beispiel demonstriert die Beziehung zwischen einer Private- und einer Feldvariablen mit den gleichen Namen. Die Private-Variable wurde .ber das MEMVAR-Statement deklariert:

```
FUNCTION Beispiel
  MEMVAR Summe, Adresse
  PRIVATE Summe := 100
  USE Kunden NEW
  //
  ? Summe          // Verweist auf Speichervariable
  ? Kunden->Summe   // Verweist auf Feldvariable
  //
RETURN NIL
```

See Also:

ALLTRIM() Entfernt f.hrende und nachfolgende Leerzeichen in einer Zeichenfolge

---

Syntax

ALLTRIM(<cZeichenfolge>) -> cTrimmZeichenfolge

Parameter

<cZeichenfolge> ist die Zeichenfolge, die bearbeitet werden soll.

R.ckgabe

ALLTRIM() gibt eine Zeichenfolge zur.ck, aus der f.hrende und nachfolgende Leerzeichen entfernt worden sind.

Anwendung

ALLTRIM() ist eine Zeichenkettenfunktion, die in einer Zeichenkette alle f.hrenden und nachfolgenden Leerzeichen entfernt. Sie ist mit der Kombination von LTRIM() und RTRIM() identisch, welche die f.hrenden und nachfolgenden Leerzeichen separat entfernen. Die entgegengesetzten Funktionen sind PADC(), PADL() und PADR(), welche Zeichenketten mit F.llzeichen beidseitig, links oder rechts, auff.llen.

Beispiele

Das Beispiel erzeugt eine Zeichenkette mit beidseitigen Leerzeichen, die in einem weiteren Schritt mit ALLTRIM() entfernt werden:

```
cFolge = SPACE(10) + "Zeichenkette" + SPACE(10)
? LEN(cFolge)                    // Resultat: 32
? LEN(ALLTRIM(cFolge))           // Resultat: 12
```

Dateien

Definitionsdatei ist STD.CH

See Also:

PARAMETERS
  Erzeugt Parameter-Variablen vom Typ private

Syntax
  PARAMETERS <idParameter Liste>

Parameter
  <idParameter Liste> sind ein oder mehrere durch Kommata getrennte
  Parameter-Variablen, mit denen .bergebene Argumente .bernommen werden
  sollen. Die Anzahl der entgegengenommenen Parameter mu. nicht unbedingt
  mit der Anzahl der von einer Prozedur oder anwenderdefinierten Funktion
  .bergebenen Argumente .bereinstimmen.

Anwendung
  Das PARAMETERS-Statement erzeugt Variablen vom Typ private, welche die
  an eine Prozedur oder anwenderdefinierten Funktion .bergebenen Werte
  entgegennehmen. Die Werte oder Referenzen, die in einem solchen Fall
  .bernommen werden, korrespondieren mit den Argumenten, welche an diese
  Routine .bergeben wurden.

  Wenn eine PARAMETERS-Anweisung ausgef.hrt wird, werden alle in der
  Parameterliste aufgef.hrten Variablen als Variablen vom Typ private
  angelegt. Eventuell bestehende PRIVAT- oder PUBLIC-Variablen .bergeordneter
  Ebenen, die .ber gleiche Namen verf.gen, werden so lange verborgen, bis die
  aktuelle Prozedur oder anwenderdefinierte Funktion .ber RETURN verlassen
  wird.

  Ein PARAMETERS-Statement ist eine ausf.hrbare Anweisung und kann daher
  an beliebiger Stelle innerhalb der aktuellen Prozedur oder anwender-
  definierten Funktion stehen. Einzige Bedingung ist, da. sie nach compile-
  time-Variablendeklarationen wie FIELD, LOCAL, MEMVAR oder STATIC angegeben
  wird.

  Parameter k.nnen auch als lokale Variablen deklariert werden, indem sie
  innerhalb einer FUNCTION- oder PROCEDURE-Deklaration in Klammern
  aufgef.hrt werden. Parameter, die auf diesem Wege deklariert wurden,
  bezeichnet man als formale Parameter. Beachten Sie jedoch bitte, da.
  nicht beide Methoden gleichzeitig in einer Prozedur oder
  anwenderdefinierten Funktion verwendet werden k.nnen. Dies w.rde einen
  schwerwiegenden Compilierfehler hervorrufen, und es w.rde keine
  Objekt-Datei erzeugt.

  Unter Clipper darf die Anzahl der entgegengenommenen Parameter von der
  Anzahl der .bergebenen Argumente abweichen. Wurden mehr Argumente
  angegeben als Parameter entgegengenommen, werden alle restlichen
  ignoriert. Werden hingegen mehr Parameter abgeholt als Argumente
  angegeben waren, so werden diese mit NIL vorbelegt. Einzelne, zu
  .bergebende Argumente, k.nnen innerhalb der Argumentenliste weggelassen
  werden. Ein Parameter, der nicht .bergeben wurde, ist immer mit NIL
  vorinitialisiert. Die Anzahl der tats.chlich .bergebenen Argumente kann
  jeweils .ber PCOUNT() bestimmt werden. PCOUNT() liefert dabei immer die
  Nummer des letzten Argumentes, .bersprungene werden also mitgez.hlt.

  Weitere Informationen .ber die Parameter.bergabe finden Sie im Abschnitt
  .ber Funktionen und Prozeduren des Kapitels Grundkonzepte innerhalb
  dieses Handbuches.

Beispiele
  . In diesem Beispiel, einer anwenderdefinierten Funktion (UDF), werden
    private-Parameter .ber das PARAMETERS-Statement entgegengenommen:

```
FUNCTION Demo
  PARAMETERS cEins, cZwei, cDrei
  ? cEins, cZwei, cDrei
RETURN NIL
```

. Dieses Beispiel ist .hnlich, .bernimmt die Parameter jedoch als lokale Variablen:

```
FUNCTION Demo( cEins, cZwei, cDrei )
   ? cEins, cZwei, cDrei
RETURN NIL
```

See Also:

PRIVATE
Deklariert und initalisiert Variablen und Arrays vom Typ private

---

Syntax
PRIVATE <Bezeichner> [[:= <Initialisierung>], ...]

Parameter
<Bezeichner> ist der Name einer Variablen oder eines Arrays, die bzw. das als PRIVATE deklariert werden soll. Folgt dem Namen <Bezeichner> ein eckiges Klammernpaar ([]), so wird die Variable als Array deklariert. Die Syntax zur Angabe einer Reihe von Arrayelementen kann wie folgt aussehen:

aArray[<nElemente1>,<nElemente2>,...] oder
aArray[<nElemente1>][<nElemente2>]...

Die maximale Anzahl von Elementen je Dimension betr.gt 4096. Die Anzahl der Dimensionen wird nur durch den verf.gbaren Speicher begrenzt.

<Initialisierung> ist eine optionale Wertzuweisung f.r eine Private-Variable. Arraybezeichnern k.nnen .ber <Initialisierung> keine Werte zugewiesen werden. Die <Initialisierung> einer Variablen mu. durch einen Zuweisungsoperator (:=), gefolgt von einem g.ltigen Clipper-Ausdruck oder einem literalen Array, erfolgen. Wird keine <Initialisierung> angegeben, erfolgt eine Vorinitialisierung der Variablen mit NIL. Bei einem Array erh.lt jedes Element den Wert NIL.

Innerhalb eines PRIVATE-Statements k.nnen eine Reihe von Variablen und Arrays angelegt und optional initialisiert werden, indem die Angaben durch Kommata getrennt werden.

Anwendung
Das PRIVATE-Statement erzeugt Variablen und Arrays, die in der aktuellen und in allen aufgerufenen Prozeduren oder anwenderdefinierten Funktionen sichtbar sind. Daher wird der G.ltigkeitsbereich f.r die Klasse dieser Variablen als dynamisch (dynamic scope) bezeichnet. Private-Variablen existieren nur f.r die Dauer der aktiven Prozedur oder bis sie explizit .ber CLEAR ALL, CLEAR MEMORY oder RELEASE gel.scht werden. Wurde eine Private-Variable oder ein Private-Array angelegt, kann eine Public-Variable mit gleichem Namen so lange nicht mehr angesprochen werden, bis die aktive Prozedur oder anwenderdefinierte Funktion verlassen wird.

Der Versuch, eine Private-Variable zu deklarieren, die zuvor mit gleichem Namen .ber FIELD, LOCAL oder STATIC deklariert wurde, ruft einen schwerwiegenden Compilierungsfehler hervor.

PRIVATE-Statements sind ausf.hrbare Anweisungen und k.nnen daher an beliebiger Stelle in einer Prozedur oder anwenderdefinierten Funktion, m.ssen jedoch nach Deklarationen wie FIELD, LOCAL, MEMVAR und STATIC angegeben werden.
Des weiteren k.nnen Private-Variablen auch durch zwei andere Methoden angelegt werden:

. Eine Wertzuweisung an eine Variable, die noch nicht existiert oder nicht sichtbar ist, erzeugt automatisch eine Private-Variable.

. Parameter, die .ber das Statement PARAMETERS entgegengenommen wurden, gleichen Private-Variablen und verf.gen .ber die gleiche Existenzdauer und Sichtbarkeit wie diese.
Die maximale Anzahl gleichzeitig existierender Private-, Public- und Array-Variablen innerhalb eines Programmes darf 2048 betragen.
Weitere Informationen .ber Variablendeklarationen und G.ltigkeitsbereiche finden Sie im Abschnitt .ber Variablen des Kapitels Grundkonzepte innerhalb dieses Handbuches.

Hinweise

. Kompatibilit.t: Die ALL-, LIKE- und EXCEPT-Bedingungen einiger dBASE-Dialekte werden von Clipper nicht unterst.tzt.

Beispiele

. Das folgende Beispiel deklariert zwei Private-Arrays und drei weitere Private-Variablen:

```
  PRIVAT aArray1[20], aArray2[10], var1, var2, var3
```

. Im n.chsten Beispiel werden mehrdimensionale Private-Arrays nach unterschiedlichen Konventionen angelegt:

```
  PRIVAT aArray1[20, 10, 5], aArray2[10][10][5]
```

. Dieses Beispiel deklariert einige Private-Variablen und initialisiert diese gleichzeitig:

```
  PRIVAT dAktuell := DATE(), cColor := SETCOLOR()
  PRIVAT aArray1 := {1, 2, 3, 4}, aArray2 := {{1, 2},{3, 4},{5, 6}}
  PRIVAT aObst := { ".pfel", "Birnen", "Kirschen" }
```

See Also:

PROCEDURE
  Deklariert den Namen einer Prozedur und deren formale Parameter
---

Syntax
```
  [STATIC] PROCEDURE <idFunktion>[(<idParameter Liste>)]
    [LOCAL                              <Bezeichner> [[:= <Initialisierung>],... ]]
    [STATIC                                                                          u<B
    [FIELD                              <idFelder Liste> [IN <idAlias>]]
    [MEMVAR                                                                          u<i
    .
    . <ausf.hrbare Anweisungen>...
    .
  [RETURN]
```

Parameter
  <idFunktion> ist der Name einer zu deklarierenden Prozedur. Der Name der
  Prozedur kann beliebig lang sein, es werden jedoch nur die ersten 10
  Zeichen unterschieden. Namen k.nnen aus einer Kombination von
  alphanumerischen Zeichen und dem Unterstrich gebildet werden. Allerdings
  d.rfen sie nicht mit einem Unterstrich beginnen, da solche Namen
  reserviert sind.

  <idParameter Liste> ist die Deklaration einer oder mehrerer Parameter-
  Variablen. Parameter-Variablen in dieser Liste sind als LOCAL angelegt.

  [STATIC] PROCEDURE deklariert eine Prozedur als statisch, so da. sie nur
  von Prozeduren und Funktionen aufgerufen werden kann, die innerhalb
  derselben Programmdatei (.PRG) stehen.

  LOCAL deklariert eine Reihe von Variablen oder Arrays als LOCAL; das
  hei.t, die Variablen sind nur f.r die Dauer der aktuellen Funktion
  existent und sichtbar. Optional k.nnen diese Variablen innerhalb der
  gleichen Anweisung mit Werten initialisiert werden.

  STATIC deklariert eine Reihe von Variablen oder Arrays als STATIC; das
  hei.t, die Variablen sind nur f.r die Dauer der aktuellen Funktion
  sichtbar, existieren jedoch f.r die Zeit der Programmausf.hrung.
  Optional k.nnen diese Variablen innerhalb der gleichen Anweisung mit
  Werten initialisiert werden.

  FIELD deklariert eine Reihe von Bezeichnern als Feldnamen. IN <idAlias>
  gibt dabei optional den Arbeitsbereich an, f.r den diese
  Feldnamensbezeichner gelten sollen.

  MEMVAR deklariert eine Reihe von Bezeichnern als PRIVATE- oder
  PUBLIC-Speichervariablen.

  RETURN verl..t die aktuelle Prozedur und .bergibt die Kontrolle wieder
  an die aufrufende Prozedur oder anwenderdefinierte Funktion zur.ck. In
  jedem Fall endet die Prozedur, wenn ein neues PROCEDURE- oder FUNCTION-
  Statement auftritt oder das Ende der Programmdatei erreicht ist.

Anwendung
  Das PROCEDURE-Statement deklariert eine Prozedur und optional eine Reihe
  von lokalen Variablen, welche die .bergebenen Parameter formal
  entgegennehmen. Eine Prozedur ist ein Unterprogramm mit darin
  enthaltenen Deklarationen und Anweisungen. Diese werden ausgef.hrt, wenn
  der Name <idFunktion> entweder gefolgt von einem Klammernpaar oder aber
  durch DO aus einem Programm heraus aufgerufen wird. Eine Prozedurdefinition
  beginnt immer mit dem PROCEDURE-Statement und endet mit dem n.chsten
  PROCEDURE- oder FUNCTION-Statement oder dem Ende der Programmdatei.

  Prozeduren dienen dazu, immer wiederkehrenden Programmcode zusammenzufassen,
  um auf ihn durch einfache Angabe des Prozedurnamens zugreifen zu k.nnen.

Prozeduren und Funktionen erh.hen die Lesbarkeit, Modularit.t, Programmpflege und Wartbarkeit. Damit helfen sie, gro.e Programme einfacher zu organisieren.

Unter Clipper sind Prozeduren das gleiche wie Funktionen mit der Ausnahme, da. Prozeduren keinen R.ckgabewert besitzen. Jede Prozedur mu. mit einem PROCEDURE-Statement beginnen und kann optional ein RETURN-Statement zum R.cksprung in das aufrufende Programm enthalten.
Prozedur-Deklarationen d.rfen jedoch nicht mit anderen Prozedur-Deklarationen verschachtelt werden.

Die Sichtbarkeit von Prozedurnamen zerf.llt in zwei Klassen. Prozeduren, die im gesamten Programm zur Verf.gung stehen sollen, bezeichnet man als Public-Prozeduren; diese werden einfach mit dem PROCEDURE-Statement deklariert. Prozeduren, die nur innerhalb eines Programmteiles (.PRG-Datei) sichtbar sein sollen, werden als Static-Prozeduren bezeichnet; sie werden .ber das STATIC PROCEDURE-Statement deklariert. STATIC PROCEDURE sagt aus, da. eine Prozedur einen Datei-weiten G.ltigsbereich (file-wide scope) besitzt.

Static-Prozeduren sind in einer Reihe von F.llen sehr hilfreich. Zum ersten begrenzen sie die Sichtbarkeit und daher auch den Zugriff auf solche Prozeduren. So k.nnen Teilprogramme innerhalb einer Programmdatei allgemein zugreifbare Prozeduren (public procedures) bereithalten, ohne Implemetierungsdetails, definiert als STATIC-Prozeduren oder -Funktionen, zu verraten. Des weiteren werden Referenzen auf static procedures bereits zur Compilierzeit aufgel.st. Dies vermeidet Konflikte mit Prozedurnamen, die erst zur Linkzeit aufgel.st werden. Damit wird sichergestellt, da. statische Prozeduren innerhalb der gleichen Programmdatei auch dann aufgerufen werden k.nnen, wenn Prozeduren mit gleichen Namen in anderen Programmdateien als Public-Prozeduren vorkommen.

Weitere Informationen .ber Prozeduren, die Variablen-Deklaration und die .bergabe von Parametern finden Sie im Kapitel Grundkonzepte.

Hinweise

. Aufruf einer Prozedur: Es bestehen zwei M.glichkeiten, eine Prozedur aufzurufen. Die erste und empfohlene Methode ist der Aufruf gleich einem Funktionsaufruf in einer einzelnen Zeile (function calling convention):

<idProcedure>([<Argumentenliste>])

Die zweite, und nicht mehr empfohlene Methode ist der Aufruf durch den Gebrauch von DO...WITH (command calling convention). Die zwei Methoden unterscheiden sich lediglich in der Art der Argumenten.bergabe. Die Funktionsaufruf-Methode .bergibt Variablen immer als Wert (per Value), w.hrend die DO...WITH-Methode diese per Referenz .bergibt.
Ein Prozedur-Aufruf kann auch mit einem Aliasbezeichner versehen werden, wenn der Aufruf nach der Funktionsaufruf-Methode erfolgt. Dabei wird der Aliasbezeichner vorangestellt und der Ausdruck in Klammern eingegrenzt:

<idAlias>->(<idProcedure>([<Argumentenliste>]))

Wird eine Prozedur in einem Alias-Ausdruck aufgerufen, gilt f.r diesen Ausdruck der Arbeitsbereich, auf den der Aliasbezeichner verweist. F.r die Dauer der Evaluierung des Ausdruckes wird also tempor.r der angegebene Arbeitsbereich selektiert. .hnlich wie bei anderen Ausdr.cken kann ein Aliasausdruck ohne Befehl in einer Zeile stehen.
Eine Prozedur kann sich in Clipper selbst aufrufen (rekursiver Aufruf). Dies bedeutet, da. innerhalb der ausf.hrbaren Anweisungen einer Prozedur ein Verweis auf dieselbe Prozedur erfolgen kann. Es ist jedoch Vorsicht geboten, da ein Laufzeitfehler (Stack-Overflow) auftritt, wenn keine geeignete Abbruchmethode vorgesehen wurde.

. Parameter: Prozeduren, wie auch anwenderdefinierte Funktionen k.nnen von dem aufrufenden Proze. (Prozedur, Funktion oder DOS-Kommandointerpreter) Parameter entgegennehmen. Ein Parameter ist ein Platzhalter f.r einen Wert oder eine Referenz. Unter Clipper 5.0 bestehen zwei M.glich-

keiten, Parameter entgegenzunehmen; entweder als Liste lokaler Variablennamen innerhalb der Deklarationszeile einer Prozedur, oder aber als Liste von Private-Variablennamen innerhalb einer PARAMETERS-Anweisung. Beachten Sie dabei bitte, da. diese Methoden nicht gemischt verwendet werden d.rfen. Dies w.rde zu einem schwerwiegenden Compilier-Fehler f.hren.

Parameter werden in der gleichen Reihenfolge entgegengenommen, wie sie auch .bergeben worden sind. Unter Clipper darf die Anzahl der entgegengenommenen Parameter von der Anzahl der .bergebenen Argumente abweichen. Einzelne zu .bergebende Argumente k.nnen innerhalb der Argumentenliste weggelassen werden. Ein Parameter, der nicht .bergeben wurde, wird mit NIL vorinitialisiert. Die Anzahl der tats.chlich .bergebenen Argumente kann jeweils .ber PCOUNT() bestimmt werden. PCOUNT() liefert dabei immer die Nummer des letzten Parameters, .bersprungene werden also mitgez.hlt.

Parameter k.nnen an eine Prozedur als Referenz (per Referenz) oder als Wert (per Value) .bergeben werden. Die Standardmethode f.r Ausdr.cke und Variablen ist abh.ngig von der Methode des jeweiligen Aufrufes. Nach der Funktionsaufruf-Methode erfolgt eine .bergabe per Value. Dies schlie.t Variablen, die eine Referenz auf Arrays oder Code-Bl.cke enthalten, ein. Nach der DO...WITH-Methode erfolgt die .bergabe per Referenz, mit der Ausnahme von Feldvariablen, die immer per Value .bergeben werden m.ssen. Die .bergabe von Feldvariablen mu. in einem Klammernpaar erfolgen, es sei denn, sie wurden zuvor mit dem FIELD-Statement deklariert. Geschieht dies nicht, tritt ein Laufzeitfehler auf.

Beispiele

. Dieses Beispiel zeigt den f.r Clipper 5.0 typischen Aufbau einer Prozedur:

```
PROCEDURE Beispiel( cName, cKlasse, nBonus )
  LOCAL nZ.hler, aSeite := ("Oben", "Unten")
  STATIC nGesamtBonus := 0
__.
  .<Ausf.hrbare Anweisungen>...
__.
RETURN
```

. Das folgende Beispiel demonstriert, wie festgestellt werden kann, ob einzelne Argumente beim Aufruf einer Prozedur weggelassen wurden:

```
PROCEDURE Test( Param1, Param2, Param3 )
  IF Param2 == NIL                  // Test des zweiten Parameters
    Param2 := "Vorgabewert"
  ENDIF
__.
  .<Anweisungen>...
__.
RETURN
```

. Dieses Beispiel zeigt, wie eine Prozedur in Verbindung mit einem Aliasbezeicher aufgerufen werden kann:

```
USE Rechnung NEW
USE Kunden NEW
//
Rechnung->(DisplValuta(EndBetrag + EndBetrag * 0.14))
```

See Also:

PUBLIC
  Deklariert und initialisiert Public-Variablen und -Arrays

Syntax
  PUBLIC <Bezeichner> [[:= <Initialisierung>], ...]

Parameter
  <Bezeichner> ist der Name einer Variablen oder eines Arrays, die bzw. das als PUBLIC deklariert werden soll. Folgt dem Namen <Bezeichner> ein eckiges Klammernpaar ([]), wird ein Array deklariert. Die Syntax zur Angabe einer Reihe von Arrayelementen kann wie folgt aussehen:

    aArray[<nElemente1>,<nElemente2>,...] oder
    aArray[<nElemente1>][<nElemente2>]...

  Die maximale Anzahl von Elementen je Dimension betr.gt 4096. Die Anzahl der Dimensionen wird nur durch den verf.gbaren Speicher begrenzt.

  <Initialisierung> ist eine optionale Wertzuweisung f.r eine Public-Variable. Arraybezeichnern k.nnen .ber <Initialisierung> keine Werte zugewiesen werden. Die <Initialisierung> einer Variablen mu. durch einen Zuweisungsoperator (:=), gefolgt von einem g.ltigen Clipper-Ausdruck oder einem literalen Array, erfolgen. Wird keine <Initialisierung> angegeben, wird die Variable mit NIL vorinitialisiert.
  Im Falle eines Arrays erh.lt jedes Element den Wert NIL.
  Innerhalb eines PUBLIC-Statements k.nnen mehrere Variablen und Arrays angelegt und optional initialisiert werden, indem die Angaben durch Kommata getrennt werden.

Anwendung
  Das PUBLIC-Statement erzeugt Variablen und Arrays, die in allen Prozeduren und anwenderdefinierten Funktionen eines Programmes sichtbar sind. Public-Variablen existieren f.r die Dauer der Programmausf.hrung oder bis sie explizit durch CLEAR ALL, CLEAR MEMORY oder RELEASE gel.scht werden. Die Deklarationen von PRIVATE-, LOCAL- oder STATIC-Variablen, die den gleichen Namen wie bereits existierende PUBLIC-Variablen besitzen, verbergen diese f.r die Dauer ihrer eigenen Sichtbarkeit. Ein Versuch, PUBLIC-Variablen anzulegen, die bereits unter dem gleichen Namen als PRIVATE-Variablen existieren und sichtbar sind, wird ignoriert (Ausnahmen entnehmen Sie bitte den weiter unten aufgef.hrten Hinweisen).

  Der Versuch, PUBLIC-Variablen zu erzeugen, deren Namen mit bereits .ber FIELD-, LOCAL- oder STATIC-Deklarationen angelegten Variablen kollidieren, erzeugt einen schwerwiegenden Compilierungsfehler; es wird also keine Objekt-Datei angelegt. Dies gilt f.r die gesamte G.ltigkeitsdauer einer Deklaration.

  PUBLIC-Statements sind ausf.hrbare Anweisungen. Deshalb k.nnen sie an beliebiger Stelle innerhalb einer Prozedur oder anwenderdefinierten Funktion, jedoch immer nach Deklarationen wie FIELD, LOCAL, MEMVAR und STATIC angegeben werden .

  Die maximale Anzahl der gleichzeitig existierenden PUBLIC- und PRIVATE-Variablen und -Arrays in einem einzelnen Programm betr.gt 2048. Weitere Informationen .ber Variablendeklarationen und deren G.ltigkeitsbereich entnehmen Sie bitte dem Abschnitt .ber Variablen aus dem Kapitel Grundkonzepte in diesem Handbuch.

Hinweise
  . PUBLIC Clipper: Damit erweiterte M.glichkeiten von Clipper genutzt werden k.nnen, ohne auf eine Kompatibilit.t zu anderen d-Sprachen verzichten zu m.ssen, wird eine Variable CLIPPER beim Programmstart mit true (.T.) vorinitialisiert, falls sie .ber PUBLIC erzeugt wurde.

  . Namenskonflikte zwischen Public-Arrays und existierenden

Private-Variablen: Das Statement PUBLIC x[10] erzeugt kein PUBLIC-Array, falls bereits eine PUBLIC- oder PRIVATE-Arrayvariable existiert. Es verwirft vielmehr den Inhalt der Variablen x und tauscht ihn gegen eine Referenz auf ein 10 Elemente umfassendes Array aus.

Beispiele

. Das folgende Beispiel deklariert zwei PUBLIC-Arrays und eine PUBLIC-Variable:

```
PUBLIC aArray1[10, 20], var1
PUBLIC aArray2[10][20]
```

. Die folgenden PUBLIC-Statements erzeugen Variablen und weisen ihnen Werte zu:

```
PUBLIC cString := SPACE(10), cColor := SETCOLOR()
PUBLIC aArray1 := { 1, 2, 3 }, aArray2 := ARRAY(10, 2)
```

See Also:

ALTD() Aufruf des Clipper-Debuggers

Syntax

ALTD([<nAktion>]) -> NIL

Parameter

<nAktion> beschreibt, welche Aktion ALTD() beim Aufruf durchf.hren soll. Im folgenden erhalten Sie eine Liste der m.glichen Aktionen:

Tabelle 5-5: ALTD() Funktionen von ALTD()

Parameter	Aktion
Keine Angabe	Ruft den Debugger auf
0	Schaltet ALT+D ab
1	Schaltet ALT+D ein und ruft den Debugger auf
Andere Angabe	Keine Aktion

R.ckgabe

ALTD() besitzt keinen R.ckgabewert.

Anwendung

Die Funktion ALTD() verh.lt sich je nach Parameter, wie oben in der Tabelle angegeben, unterschiedlich. Um mehr Informationen .ber den Debugger zu erhalten, lesen Sie bitte im Handbuch Der Clipper-Debugger nach.

Beispiele

Dieses Beispiel zeigt eine Reihe symbolischer Konstanten, die als Parameter f.r ALTD() verwendet werden k.nnen, bevor der Debugger aufgerufen wird:

```
#define DEBUG_AUS 0
#define DEBUG_AN 1
//
ALTD(DEBUG_AN)
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

RETURN
Beendet eine Prozedur, anwenderdefinierte Funktion oder ein Programm

---

Syntax
RETURN [<exp>]

Parameter
<exp> ist ein Ausdruck beliebigen Typs, der evaluiert und als Ergebniswert von einer anwenderdefinierten Funktion zur.ckgegeben wird. Falls eine anwenderdefinierte Funktion ohne RETURN-Statement beendet wurde, liefert sie NIL als R.ckgabewert.

Anwendung
RETURN beendet eine Prozedur, anwenderdefinierte Funktion oder ein Programm durch die R.ckgabe der Programmkontrolle an die aufrufende Routine. Ein RETURN aus der h.chsten Prozedurebene .bergibt die Programmkontrolle wieder an das Betriebssystem. Alle angelegten PRIVATE- und LOCAL-Variablen werden freigegeben, bevor die Kontrolle an die aufrufende Prozedur zur.ckgegeben wird.

Es kann mehr als ein RETURN-Statement in einer Prozedur oder anwenderdefinierten Funktion enthalten sein. Eine Prozedur oder anwenderdefinierte Funktion mu. nicht mit einem RETURN beendet werden. Da anwenderdefinierte Funktionen jedoch einen Wert zur.ckgeben, mu. mindestens ein RETURN-Statement mit einem Argument angegeben sein.

Hinweise
. Arrays: Da der Datentyp eines Arrays wie jeder andere Datentyp behandelt wird, kann von einer anwenderdefinierten Funktion auch ein Array als Ergebniswert .ber das RETURN-Statement zur.ckgegeben werden.

. RETURN TO MASTER: Clipper unterst.tzt kein RETURN TO MASTER oder eine andere Form von RETURN, .ber die in eine bestimmte Programmebene zur.ckgesprungen werden kann. Falls erforderlich, kann dies durch ein BEGIN SEQUENCE...END-Konstrukt realisiert werden.

Beispiele
. Dieses Beispiel zeigt die formale Form der RETURN-Statements in einer Prozedur oder anwenderdefinierten Funktion:

```
  PROCEDURE <idProzedur>
  ....
     .<Anweisungen>...
  ....
  RETURN

  FUNCTION <idFunktion>
  ....
     .<Anweisungen>...
  ....
  RETURN <expReturn>
```

. Das folgende Beispiel zeigt, wie ein Array als Ergebnis einer anwenderdefinierten Funktion zur.ckgegeben wird:

```
  FUNCTION MwstArray
     LOCAL aArray := { 14, 7, 0, 0 }
  RETURN aArray
```

See Also:

STATIC
Deklariert und initialisiert statische Variablen und Arrays

Syntax
STATIC <Bezeichner> [[:= <Initialisierung>], ...]

Parameter
<Bezeichner> ist der Name einer Variablen oder eines Arrays, die bzw. das als STATIC deklariert werden soll. Folgt dem Namen <Bezeichner> ein eckiges Klammernpaar ([]), wird ein Array deklariert. Die Syntax zur Angabe einer Reihe von Arrayelementen kann wie folgt aussehen:

```
aArray[<nElemente1>,<nElemente2>,...] oder
aArray[<nElemente1>][<nElemente2>]...
```

Die maximale Anzahl von Elementen je Dimension betr.gt 4096. Die Anzahl der Dimensionen wird nur durch den verf.gbaren Speicher begrenzt.

<Initialisierung> ist eine optionale Wertzuweisung f.r eine STATIC-Variable. Die <Initialisierung> einer Variablen mu. durch den Zuweisungs-operator (:=) erfolgen. Dabei mu. ein zur Compilierzeit feststehender, konstanter Ausdruck, bestehend aus Konstanten und Operatoren oder aus einem literalen Array, zugewiesen werden. Wird keine <Initialisierung> angegeben, wird die Variable mit NIL vorinitialisiert. Im Falle eines Arrays erh.lt jedes Element den Wert NIL.

Hinweis
Der Makro-Operator (&) darf in einer STATIC-Deklaration nicht verwendet werden.

Anwendung
Das STATIC-Statement deklariert Variablen und Arrays, die bereits bei Programmstart existieren und deren Lebensdauer f.r die Laufzeit eines Programmes gilt, die aber nur in den Programmteilen, in denen sie deklariert wurden, sichtbar sind. Die Sichtbarkeit statischer Variablen erstreckt sich also jeweils auf Prozeduren und Funktionen, in denen sie nach einem PROCEDURE- oder FUNCTION-Statement deklariert wurden.

STATIC-Variablen sind f.r alle Funktionen und Prozeduren innerhalb einer Programmdatei sichtbar, wenn sie dort vor den FUNCTION- oder PROCEDURE-Statements angegeben werden und au.erdem bei der Compilation des Programmes die Option /N angegeben worden ist.

Alle statischen Variablen werden angelegt, sobald das Programm aufgerufen wird. Die Werte einer <Initialisierung> werden zugewiesen, bevor die Ausf.hrung des Programmes beginnt.

Die Deklaration statischer Variablen mu. vor einer PRIVATE-, PUBLIC-, PARAMETERS- oder anderen ausf.hrbaren Deklaration erfolgen. Eine STATIC-Variable, die bereits mit gleichen Namen im Kopf einer FUNCTION- oder PROCEDURE-Deklaration als FIELD-, LOCAL- oder MEMVAR-Variable deklariert ist, erzeugt einen schwerwiegenden Fehler bei der Compilierung des Programmes.

Die maximale Anzahl statischer Variablen ist in einem Programm nur durch den zur Verf.gung stehenden Speicher begrenzt. Die als STATIC deklarierten Arrays d.rfen je Dimension maximal 4096 Elemente enthalten, wobei die Anzahl der Dimensionen nicht begrenzt ist.

Hinweise
. .berpr.fung lokaler Variablen innerhalb des Debuggers: Sollen innerhalb des Clipper-Debuggers statische Variablen ansprechbar sein, m.ssen die Programmdateien (.PRG) mit dem Compilerschalter /B compiliert werden, damit die dazu ben.tigte Information in die Objekt-Datei (.OBJ)

aufgenommen wird.

. Makro-Ausdr.cke: Statische Variablen k.nnen nicht innerhalb von Makro-Variablen oder Ausdr.cken angegeben werden. Existieren PRIVATE- oder PUBLIC-Variablen mit dem gleichen Namen wie statische Variablen, werden diese in den Makro-Ausdr.cken verwendet. Existieren solche Variablen nicht, kommt es zu einem Laufzeitfehler.

. MEMORY-Dateien: Statische Variablen k.nnen nicht in Memory-Dateien (.MEM) gespeichert oder aus solchen entnommen werden.

. Datentyp statischer Variablen: Die Funktion TYPE() verwendet einen Makro-Operator (&), um den Typ eines .bergebenen Argumentes zu ermitteln. Diese Funktion kann daher nicht zur Ermittlung des Typs von statischen oder lokalen Variablen herangezogen werden. Verwenden Sie stattdessen die Funktion VALTYPE().

Beispiele

. Das folgende Beispiel deklariert statische Variablen und weist ihnen zum Teil Initialisierungswerte zu:

```
  STATIC aArray1[20, 10], aArray2[20][10]
  STATIC var1, var2
  STATIC nZaehler := 0
  STATIC aArray := { 1, 2, 3 }
```

. Die folgende Funktion zeigt die Manipulation einer statischen Variable:

```
  FUNCTION Zaehler( nWertNeu )
    STATIC nZaehlWert := 0
    IF nWertNeu <> NIL
      nZaehlWert := nWertNeu
    ELSE
      nZaehlwert++
    ENDIF
  RETURN nZaehlWert
```

. Das n.chste Beispiel zeigt die Verwendung einer statischen Variablen innerhalb der gesamten Datei:

```
  STATIC aArray := { 1, 2, 3, 4, 5 }

  FUNCTION Eins
    ? aArray[1]
  RETURN NIL

  FUNCTION Zwei
    ? aArray[2]
  RETURN NIL
```

See Also:

Das Extend-System

Das Extend-System bildet eines der Schl.sselelemente in der offenen Architektur von Clipper. Es erlaubt Ihnen, Funktionen in C oder Assembler zu schreiben, die, genau wie jede in Clipper geschriebene UDF, direkt in Ihrem Programm aufgerufen werden k.nnen.

Dieses Erweiterungssystem besteht aus zwei Funktionsbl.cken, die in der Bibliothek CLIPPER.LIB definiert sind. Der eine Block stellt die Schnittstelle zu C, der andere die zu Assembler zur Verf.gung, in beiden Bl.cken existieren Funktionen, um Daten von Clipper zu .bernehmen, Daten an Clipper zur.ckzugeben und Daten direkt in Clipper-Variablen zu speichern. Die Arbeit mit diesen Schnittstellen-Funktionen wird durch die mitgelieferten Makros zus.tzlich erleichtert.

Das Protokoll des Extend-Systems

Das Extend-System besteht aus verschiedenen, zusammenh.ngenden Elementen:

Laufzeit-Datenstrukturen: Interne Datenstrukturen, die es einem compilierten Clipper-Programm erlauben, Parameter und R.ckgabewerte zu verwalten.

Schnittstellenfunktionen: Funktionen aus CLIPPER.LIB, die es C- und Assembler-Routinen erlauben, die internen Datenstrukturen anzusprechen und zu manipulieren.

Servicefunktionen: Funktionen aus CLIPPER.LIB, die zur Unterst.tzung von C- und Assembler-Routinen dienen, so zum Beispiel Funktionen zur Speicherverwaltung.

Header-Dateien und Definitionen: Die im Lieferumfang von Clipper enthaltenen Header-Dateien, sowie die darin getroffenen Definitionen, sollten in allen C- und Assembler-Funktionen eingesetzt werden.

Der vielleicht wichtigste Teil des Extend-Systems ist das sogenannte Extend-Protokoll, das Richtlinien f.r den Einsatz des Extend-Systems definiert. Hierbei werden die folgenden Spezifikationen getroffen:

. Namensgebung f.r Funktionen
. Speichermodell
. Aufrufkonventionen
. Datentypen und Konvertierungen
. Zul.ssige Operationen
. Die Namen und Parameter der Schnittstellen- und Servicefunktionen

Namensgebung f.r Funktionen

Der Aufruf von C- oder Assembler-Routinen aus Clipper heraus bedeutet zwangsl.ufig, da. zwei oder mehr Objektdateien zusammen gelinkt werden m.ssen. Einige dieser Dateien enthalten Clipper-Code, die anderen C- bzw. Assembler-Routinen.

W.hrend der Compilation erzeugt Clipper eine Liste aller referierten Funktions- oder Prozeduraufrufe, die im Sourcetext angesprochen, aber nicht definiert werden. Diese Referenzen werden externe Referenzen genannt, und von Clipper mit in die Objektdatei geschrieben. W.hrend des Linkvorganges benutzt der Linker diese Namen, um in anderen Objektdateien oder Bibliotheken nach Aufl.sungen dieser Referenzen, also nach einem globalen Symbol des gleichen Namens, zu suchen. Wird ein solches Symbol gefunden, bindet der Linker das definierte Modul mit in die .EXE Datei ein, andernfalls wird ein Linkfehler erzeugt.

Falls C- oder Assembler-Module in eine Clipper-Applikation eingebunden werden sollen, mu. sorgf.ltig auf die Namensgebung f.r diese Routinen geachtet werden, damit der Linker die Referenzen im Clipper-Programm

korrekt aufl.sen kann.

. Funktions- und Prozedurnamen m.ssen unter Clipper mit einem Buchstaben oder Unterstrich beginnen; sie d.rfen nur Buchstaben, Ziffern und Unterstriche enthalten. Assembler hat zwar weniger strenge Namensvorschriften, trotzdem m.ssen sich Funktionen, die aus Clipper heraus aufgerufen werden sollen, an die Clipper-Namenskonventionen halten.

. F.r Clipper sind nur die ersten zehn Zeichen eines Funktions- oder Prozedurnamens signifikant, l.ngere Namensteile werden w.hrend der Compilation abgeschnitten. Sowohl C als auch Assembler erlauben normalerweise l.ngere Funktionsnamen, man sollte den aus Clipper aufzurufenden Routinen jedoch immer nur Namen mit zehn oder weniger Zeichen geben.

. C und Assembler behandeln Funktionsnamen Case-sensitiv, Funktionsnamen mit Kleinbuchstaben werden also von gleichnamigen, gro.geschriebenen Funktionen unterschieden. Clipper arbeitet Case-insensitiv, alle Funktionsnamen werden in Gro.buchstaben umgewandelt. Funktionen, die aus Clipper aufgerufen werden, sollten daher Funktionsnamen erhalten, die nur aus Gro.buchstaben bestehen.

. Viele C-Compiler erzeugen bei globalen Symbolen einen Unterstrich vor dem Symbolnamen. Bei den Funktionen, die aus Clipper heraus aufgerufen werden, mu. dieser Unterstrich unterdr.ckt werden.

Die mitgelieferte Header-Datei EXTEND.H enth.lt eine #define-Direktive CLIPPER, die mit dem Microsoft C-Compiler eingesetzt werden kann, um Funktionsnamen in Gro.buchstaben ohne f.hrenden Unterstrich zu .bersetzen.

Achtung

.RTLink und die meisten anderen Linker arbeiten normalerweise im Case-insensitiv Modus, es sei denn, eine spezielle Linkoption (bei .RTLink ist das die /NOIGNORECASE-Option) wird eingesetzt. Es geh.rt dennoch zur besseren Programmierpraxis, die entsprechenden Funktionsnamen bereits im Quelltext gro.zuschreiben.

Speichermodell

Die Prozessorfamilie des Intel 8086 benutzt die sogenannte Segmentadressierung. Dadurch werden Programme in Speichersegmente unterteilt, die jeweils bis zu 64 KB gro. sein k.nnen. Speicheradressen bestehen aus zwei Teilen, einer Segmentnummer und einem Offset innerhalb des Segmentes. Spricht ein Programm eine bestimmte Speicherstelle an, wird die entsprechende Segmentnummer in das Segmentregister geladen. Danach k.nnen verschiedene Offsets benutzt werden, um die Daten innerhalb des Segmentes zu adressieren.

Der Ausdruck Speichermodell bezeichnet die Organisation der Speichersegmente in einem Programm. Mit small model werden Programme bezeichnet, die komplett in ein oder zwei Segmente passen. Bei diesen Programmen werden die Segmentregister bereits bei Programmstart geladen und behalten ihre Werte w.hrend des kompletten Programmlaufes bei. Dadurch kann eine Speicheradresse nur durch einen Offset bezeichnet werden. Diese einfache Offsetadressierung wird auch near Adressierung genannt.

_parc() .bernahme eines Pointers auf eine Clipper-Zeichenkette

---

C-Prototyp

```
#include "extend.h"
char * _parc(int paramNum [, int arrayIndex])
```

Parameter

paramNum ist die Position des Parameters in der aktuellen Parameterliste.

arrayIndex ist ein Array-Index, um ein bestimmtes Element anzusprechen, falls ein Array .bergeben wurde.

R.ckgabe

_parc() gibt einen far-Pointer auf eine Zeichenkette zur.ck.

Anwendung

_parc() wird benutzt, um einen Pointer auf einen String zu erzeugen, der von Clipper .bergeben wurde. Dieser Zeiger kann, mu. aber nicht auf den aktuellen Clipper-Variableninhalt oder auf eine Kopie des Wertes zeigen. M.glicherweise zeigt er auf einen Speicherbereich, der von mehreren Clipper-Variablen angesprochen wird.

Daher sollte der durch _parc() erhaltene Pointer nicht zur direkten Ver.nderung des Character-Wertes eingesetzt werden, da sonst unvorhersehbare Ergebnisse die Folge sein k.nnen. Um einen Character-Wert zu ver.ndern, sollte eine bearbeitete Kopie .ber _retc() an Clipper zur.ckgegeben werden. Alternativ kann der Character-Wert auch per Referenz .bergeben und dann mit _storc() direkt ver.ndert werden.

Beispiele

In C:

```
char *str;
str = _parc(1);
```

In Assembler:

```
EXTRN   __parc:FAR
  mov   ax, 1
  push  ax
  call  __parc       ; Pointer in DX:AX
  add   sp, 2        ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

ARRAY() Erzeugt ein uninitialisiertes Array in der angegebenen L.nge

---

Syntax

ARRAY(<nL.nge> [, <nL.nge>...]) -> aArray

Parameter

<nL.nge> ist die Anzahl der Elemente einer Dimension. Die maximale Anzahl von Elementen betr.gt 4096. Die Anzahl der Dimensionen in Clipper sind unbegrenzt.

R.ckgabe

ARRAY() liefert ein Array in der angegebenen Dimension.

Anwendung

ARRAY() ist eine Arrayfunktion, welche ein uninitialisiertes Array mit einer angegebenen Anzahl von Elementen und Dimensionen zur.ckgibt. Wird mehr als ein Parameter <nL.nge> angegeben, wird ein mehrdimensionales Array erzeugt, bei dem die Anzahl der Dimensionen gleich der Anzahl der Parameter <nL.nge> ist.

In Clipper 5.0 bestehen mehrere M.glichkeiten, ein Array zu erzeugen. Ein Array kann durch die Deklarationsanweisungen LOCAL oder STATIC, durch die Anweisung PRIVAT oder PUBLIC, durch literale Zuweisung oder durch den Gebrauch der Funktion ARRAY() erzeugt werden. ARRAY() hat den Vorteil, da. Arrays innerhalb von Ausdr.cken oder Code-Bl.cken angelegt werden k.nnen.

Beispiele

. Dieses Beispiel erzeugt ein eindimensionales Array mit f.nf Elementen durch den Gebrauch von ARRAY() und zeigt eine vergleichbare Anweisung durch die Zuweisung literaler Werte:

```
aArray  := ARRAY(5)
aArray  := { NIL, NIL, NIL, NIL, NIL }
```

. Dieses Beispiel zeigt drei verschiedene Methoden, welche das gleiche, mehrdimensionale Array erzeugen:

```
aArray  := ARRAY( 3, 2)
aArray  := {{ NIL, NIL },{ NIL, NIL },{ NIL, NIL }}
aArray  := { ARRAY(2), ARRAY(2), ARRAY(2)}
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

_parclen() Ermittlung der L.nge einer .bergebenen Zeichenkette

---

C-Prototyp

```
#include "extend.h"
int _parclen(int paramNum [, int arrayIndex])
```

Parameter

paramNum ist die Position des Parameters in der aktuellen Parameterliste.

arrayIndex ist ein Array-Index, um ein bestimmtes Element anzusprechen, falls ein Array .bergeben wurde.

R.ckgabe

_parclen() gibt die L.nge der Zeichenkette als unsigned int zur.ck.

Anwendung

_parclen() wird benutzt, um die L.nge eines von Clipper .bergebenen Strings festzustellen. Dabei werden im String enthaltene NULL-Bytes (ASCII 0) mitgez.hlt, der abschlie.ende NULL-Terminator jedoch nicht.

Beispiele

In C:

```
unsigned int len;
len = _parclen(1);
```

In Assembler:

```
EXTRN   __parclen:FAR
  mov   ax, 1
  push  ax
  call  __parclen   ; L.nge in AX
  add   sp, 2       ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_pards() .bernahme eines Pointers auf einen Clipper-Datumsstring

---

C-Prototyp

```
#include "extend.h"
char *_pards(int paramNum [, int arrayIndex])
```

Parameter

paramNum ist die Position des Parameters in der aktuellen Parameterliste.

arrayIndex ist ein Array-Index, um ein bestimmtes Element anzusprechen, falls ein Array .bergeben wurde.

R.ckgabe

_pards() gibt einen far-Pointer auf eine Zeichenkette zur.ck, die den Datumswert darstellt.

Anwendung

_pards() wird benutzt, um einen Datumsparameter aus Clipper zu .bernehmen. Wenn Sie _pards() benutzen, wird das von Clipper .bergebene Datum in eine Zeichenkette der Form JJJJMMTT konvertiert. Der Zeiger auf diesen String wird zur.ckgegeben.

_pards() benutzt einen statisch allozierten Buffer, der bei jedem Funktions-Aufruf .berschrieben wird. Sie m.ssen daher eine Kopie des jeweiligen Wertes anlegen, um mehr als einen Datumswert aus Clipper zu .bernehmen.

Beispiele

In C:

```
char *str;
str = _pards(1);
```

In Assembler:

```
EXTRN   __pards:FAR
  mov   ax, 1
  push  ax
  call  __pards        ; Pointer in DX:AX
  add   sp, 2          ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_parinfa() Ermittlung des Typs eines Arrayelementes

C-Prototyp

```
#include "extend.h"
int _parinfa(int paramNum [, int arrayIndex])
```

Parameter

paramNum ist die Position des Parameters in der aktuellen Parameterliste.

arrayIndex ist ein Array-Index, um ein bestimmtes Element anzusprechen, falls ein Array .bergeben wurde.

R.ckgabe

Wenn der Parameter arrayIndex 0 ist, wird die Gr..e des Arrays in AX zur.ckgegeben. Hat der Parameter arrayIndex einen Wert gr..er 0, wird in AX der Typ des Elementes zur.ckgegeben. Die R.ckgabewerte sind in der folgenden Tabelle zusammengefa.t:

Anwendung

_parinfa() gibt den Typ eines beliebigen Arrayelementes zur.ck und wird zur Parameterpr.fung benutzt. Da Clipper-Arrayelemente unterschiedlichen Typs sein d.rfen, sollten Sie jedes Arrayelement explizit auf seinen Typ pr.fen, bevor Sie es in Ihrer Routine ansprechen.

Beachten Sie bitte, da. _parinfa(paramNum, 0) die Anzahl der Elemente im .bergebenen Array ermittelt. Diesen Wert erh.lt man auch mit Hilfe des Makros ALENGTH() aus EXTEND.H.

Beispiele

In C:

```
int len;
int type;
len = _parinfa(1, 0);
type = _parinfa(1, 1);
```

In Assembler:

```
EXTRN   __parinfa:FAR
  mov   ax, 0
  push  ax
  mov   ax, 1
  push  ax
  call  __parinfa       ; L.nge in AX
  add   sp, 4           ; Stack zur.cksetzen

  mov   ax, 1
  push  ax
  push  ax
  call  __parinfa       ; Typ Konstante in AX
  add   sp, 4           ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_parinfo() Ermittlung des Typs eines Parameters

C-Prototyp

```
#include "extend.h"
int _parinfo(int paramNum)
```

Parameter

paramNum ist die Position des Parameters in der aktuellen Parameterliste.

R.ckgabe

Wenn der Parameter paramNum 0 ist, wird die Anzahl der .bergebenen Parameter zur.ckgegeben. Hat paramNum einen Wert gr..er 0 wird der Datentyp des Parameters zur.ckgegeben. Die R.ckgabewerte sind in der folgenden Tabelle zusammengefa.t:

Anwendung

_parinfo() gibt den Typ eines beliebigen Parameters zur.ck und wird zur Parameterpr.fung benutzt. Beachten Sie bitte, da. _parinfo(0) die Anzahl der .bergebenen Parameter ermittelt und damit denselben Wert zur.ckgibt wie das PCOUNT-Makro.

Um festzustellen, ob ein Parameter per Referenz .bergeben wurde, mu. ein logisches OR mit dem urspr.nglichen Typ und MPTR ausgef.hrt werden. Wenn z.B. ein String per Referenz .bergeben wurde, gibt _parinfo(n) den Wert 33 zur.ck (32 OR 1).

Das Makro ISBYREF() in EXTEND.H kann ebenfalls eingesetzt werden, um festzustellen, ob ein Parameter per Referenz .bergeben wurde. Diese Pseudofunktion bekommt die Parameternummer als Argument und gibt .T. zur.ck, wenn dieser Parameter per Referenz .bergeben wurde.

Beispiele

In C:

```
int count;
int type;
count = _parinfo(0);
type = _parinfo(1);
```

In Assembler:

```
EXTRN   __parinfo:FAR
  mov   ax, 0
  push  ax
  call  __parinfo          ; Anzahl in AX
  add   sp, 2              ; Stack zur.cksetzen

  mov   ax, 1
  push  ax
  call  __parinfo          ; Typ Konstante in AX
  add   sp, 2              ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_parl() .bernimmt einen logischen Parameter als Integer

C-Prototyp

```
#include "extend.h"
int _parl(int paramNum [, int arrayIndex])
```

Parameter

paramNum ist die Position des Parameters in der aktuellen Parameterliste.

arrayIndex ist ein Array-Index, um ein bestimmtes Element anzusprechen, falls ein Array .bergeben wurde.

R.ckgabe

_parl() gibt einen C-Integer zur.ck, der den logischen Wert darstellt, 1 f.r .T. und 0 f.r .F.

Anwendung

_parl() wird benutzt, um logische Clipper-Parameter zu .bernehmen.

Beispiele

In C:

```
int log;
log = _parl(1);
```

In Assembler:

```
EXTRN   __parl:FAR
  mov   ax, 1
  push  ax
  call  __parl          ; R.ckgabe in AX
  add   sp, 2           ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_parnd() .bernimmt einen numerischen Parameter als Double

C-Prototyp

```
#include "extend.h"
double _parnd(int paramNum [, int arrayIndex])
```

Parameter

paramNum ist die Position des Parameters in der aktuellen Parameterliste.

arrayIndex ist ein Array-Index, um ein bestimmtes Element anzusprechen, falls ein Array .bergeben wurde.

R.ckgabe

_parnd() gibt den Wert des spezifizierten Parameters als C-Double zur.ck.

Anwendung

Die _parnd() Funktion liefert einen 8-Byte (IEEE) Flie.komma-Wert. Dieses Format reicht aus, um jeden in Clipper darstellbaren Wert zu .bernehmen. Wird _parnd() von Assembler aus aufgerufen, gibt die Funktion einen Pointer in DX:AX zur.ck. Dieser Pointer zeigt auf einen statischen Puffer, der eine Kopie des Wertes enth.lt. Dieser Puffer wird bei jedem Aufruf von _parnd() .berschrieben, die Zahl sollte also f.r die weitere Verwendung kopiert werden. In C wird der Pufferinhalt automatisch in die C-Variable kopiert.

Beispiele

In C:

```
double num;
num = _parnd(1);
```

In Assembler:

```
EXTRN   __parnd:FAR
  mov   ax, 1
  push  ax
  call  __parnd
  add   sp, 2          ; Stack zur.cksetzen
  mov   es, dx         ; Pointer in ES:BX laden
  mov   bx, ax         ; Nun Wert sichern
  mov   word ptr MySave, es:[bx]
  mov   word ptr MySave2, es:[bx2]
  mov   word ptr MySave4, es:[bx4]
  mov   word ptr MySave6, es:[bx6]
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_parni() .bernimmt einen numerischen Parameter als Integer

C-Prototyp

```
#include "extend.h"
int _parni(int paramNum [, int arrayIndex])
```

Parameter

paramNum ist die Position des Parameters in der aktuellen Parameterliste.

arrayIndex ist ein Array-Index, um ein bestimmtes Element anzusprechen, falls ein Array .bergeben wurde.

R.ckgabe

_parni() gibt den Wert des spezifizierten Parameters als C-Integer zur.ck.

Anwendung

_parni() wird zur .bernahme eines numerischen Clipper-Parameters benutzt. Die Funktion konvertiert den Wert in einen C-Integer.

Beispiele

In C:

```
int num;
num = _parni(1);
```

In Assembler:

```
EXTRN   __parni:FAR
  mov   ax, 1
  push  ax
  call  __parni          ; Wert in AX
  add   sp, 2            ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_parnl() .bernimmt einen numerischen Parameter als Long Integer

---

C-Prototyp

```
#include "extend.h"
int _parnl(int paramNum [, int arrayIndex])
```

Parameter

paramNum ist die Position des Parameters in der aktuellen Parameterliste.

arrayIndex ist ein Array-Index, um ein bestimmtes Element anzusprechen, falls ein Array .bergeben wurde.

R.ckgabe

_parnl() gibt den Wert des spezifizierten Parameters als C-Long Integer zur.ck.

Anwendung

_parnl() wird zur .bernahme eines numerischen Clipper-Parameters benutzt. Die Funktion konvertiert den Wert in einen C-Long Integer.

Beispiele

In C:

```
long num;
num = _parnl(1);
```

In Assembler:

```
EXTRN   __parnl:FAR
  mov   ax, 1
  push  ax
  call  __parnl          ; Wert in DX:AX
  add   sp, 2            ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_ret() Gibt einen NIL-Wert an Clipper zur.ck

C-Prototyp

```
#include "extend.h"
void _ret(void)
```

R.ckgabe

_ret() hat keinen R.ckgabewert.

Anwendung

_ret() kopiert einen NIL-Wert in den R.ckgabebereich von Clipper. Dieser Wert ist unter Clipper als R.ckgabewert der C- oder Assembler-Routine verf.gbar, sobald die Routine die Ablaufkontrolle zur.ckgegeben hat.

Beispiele

In C:

```
_ret();
```

In Assembler:

```
EXTRN _ret:FAR
  call  _ret
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_retc() Gibt einen String als Zeichenkette an Clipper zur.ck

C-Prototyp

```
#include "extend.h"
void _retc(char *string)
```

Parameter

string ist ein far-Pointer auf eine Zeichenfolge, die als Typ character zur.ckgegeben wird.

R.ckgabe

_retc() hat keinen R.ckgabewert.

Anwendung

_retc() kopiert einen String als Clipper-Zeichenkette in den R.ckgabebereich von Clipper. Dieser Wert ist unter Clipper als R.ckgabewert der C- oder Assembler-Routine verf.gbar, sobald die Routine die Ablaufkontrolle zur.ckgegeben hat. _retc() durchsucht den String nach dem ersten NULL-Byte (ASCII 0), um so die logische L.nge der Zeichenkette festzustellen. Sollen bin.re Daten, die ein oder mehrere NULL-Bytes enthalten, zur.ckgegeben werden, mu. _retclen() benutzt werden.

Achtung

_retc() alloziert Speicher im Clipper-Heap und erzeugt eine Kopie des .bergebenen Parameters. Der String selbst mu. nicht weiter existieren, sobald _retc() beendet wurde.

Beispiele

In C:

```
_retc("Hallo Welt");
```

In Assembler:

```
EXTRN   __retc:FAR
  mov   dx, seg MyString    ; Adresse des Strings
  mov   ax, offset MyString
  push  dx
  push  ax
  call  __retc
  add   sp, 4           ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_retclen() Gibt einen String mit expliziter L.nge als Zeichenkette an Clipper zur.ck

---

C-Prototyp

```
#include "extend.h"
void _retclen(char *string, unsigned int length)
```

Parameter

string ist ein far-Pointer auf die Daten, die als Character-Wert zur.ckgegeben werden sollen.

length gibt die L.nge der Zeichenfolge an.

R.ckgabe

_retclen() hat keinen R.ckgabewert.

Anwendung

_retclen() kopiert einen String als Clipper-Zeichenkette in den R.ckgabebereich von Clipper. Dieser Wert ist unter Clipper als R.ckgabewert der C- oder Assembler-Routine verf.gbar, sobald die Routine die Ablaufkontrolle zur.ckgegeben hat.

Da _retclen() die L.nge des Strings als explizites Argument erh.lt, mu. der String nicht durch ein NULL-Byte (ASCII 0) abgeschlossen sein. Der String kann auch mehrere NULL-Bytes beinhalten.

Achtung

_retclen() alloziert Speicher im Clipper-Heap und erzeugt eine Kopie des .bergebenen Parameters. Der String selbst mu. nicht weiter existieren, sobald _retclen() beendet wurde.

Beispiele

In C:

```
char data[5] = {1, 0, 2, 0, 3, 5}
_retclen(data,5)
```

In Assembler:

```
EXTRN   __retclen:FAR
  mov   ax, 5
  push  ax
  mov   dx, seg MyString
  mov   ax, offset MyString
  push  dx
  push  ax
  call  __retclen
  add   sp, 6           ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_retds() Gibt einen String als Datumswert an Clipper zur.ck

C-Prototyp

```
#include "extend.h"
void _retds(char *string)
```

Parameter

string ist ein far-Pointer auf einen null-terminierten String, der den Datumswert in der Form JJJJMMTT repr.sentiert.

R.ckgabe

_retds() hat keinen R.ckgabewert.

Anwendung

_retds() kopiert einen Datumswert in den R.ckgabebereich von Clipper. Dieser Wert ist unter Clipper als R.ckgabewert der C- oder Assembler-Routine verf.gbar, sobald die Routine die Ablaufkontrolle zur.ckgegeben hat.

_retds() konvertiert den .bergebenen C-String in einen Clipper-Datumswert. Wird ein String .bergeben, der ein ung.ltiges Datum darstellt, erzeugt _retds() einen leeren Datumswert.

Achtung

_retds() konvertiert den .bergebenen Parameter. Der String selbst mu. nicht weiter existieren, sobald _retds() beendet wurde.

Beispiele

In C:

```
_retds("19901231")
```

In Assembler:

```
EXTRN   __retds:FAR
  mov   dx, seg MyDateString
  mov   ax, offset MyDateString
  push  dx
  push  ax
  call  __retds
  add   sp, 4           ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_retl() Gibt einen Integer als logischen Wert an Clipper zur.ck

---

C-Prototyp

```
#include "extend.h"
void _retl(int logical)
```

Parameter

logical ist der Integer, der den logischen Wert darstellt. Eine 0 wird zu .F., jeder andere Wert zu .T. konvertiert.

R.ckgabe

_retl() hat keinen R.ckgabewert.

Anwendung

_retl() kopiert einen logischen Wert in den R.ckgabebereich von Clipper. Dieser Wert ist unter Clipper als R.ckgabewert der C- oder Assembler-Routine verf.gbar, sobald die Routine die Ablaufkontrolle zur.ckgegeben hat.

_retl() interpretiert 0 als .F., jeden anderen Wert als .T.

Beispiele

In C:

```
_retl(1);
```

In Assembler:

```
EXTRN   __retl:FAR
  mov   ax, 1
  push  ax
  call  __retl
  add   sp, 2          ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_retnd() Gibt einen Double-Wert als numerischen Wert an Clipper zur.ck

---

C-Prototyp

```
#include "extend.h"
void _retnd(double n)
```

Parameter

n ist ein numerischer Ausdruck von Typ Double.

R.ckgabe

_retnd() hat keinen R.ckgabewert.

Anwendung

_retnd() kopiert einen numerischen Wert in den R.ckgabebereich von Clipper. Dieser Wert ist unter Clipper als R.ckgabewert der C- oder Assembler-Routine verf.gbar, sobald die Routine die Ablaufkontrolle zur.ckgegeben hat.

Beispiele

In C:

```
_retnd( (double) 3.14 );
```

In Assembler:

```
EXTRN     __retnd:FAR
  push    word prt (MyPi6)
  push    word prt (MyPi4)
  push    word prt (MyPi2)
  push    word prt MyPi
  call    __retnd
  add     sp, 8           ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_retni() Gibt einen Integer als numerischen Wert an Clipper zur.ck

---

C-Prototyp

```
#include "extend.h"
void _retni(int n)
```

Parameter

n ist ein numerischer Ausdruck vom Typ Integer.

R.ckgabe

_retni() hat keinen R.ckgabewert.

Anwendung

_retni() kopiert einen numerischen Wert in den R.ckgabebereich von Clipper. Dieser Wert ist unter Clipper als R.ckgabewert der C- oder Assembler-Routine verf.gbar, sobald die Routine die Ablaufkontrolle zur.ckgegeben hat.

Beispiele

In C:

```
_retni( 99 );
```

In Assembler:

```
EXTRN   __retni:FAR
  mov   ax, 99
  push  ax
  call  __retni
  add   sp, 2          ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_retnl() Gibt einen Long Integer als numerischen Wert an Clipper zur.ck

---

C-Prototyp

```
#include "extend.h"
void _retnl(long n)
```

Parameter

n ist ein numerischer Ausdruck von Typ Long Integer.

R.ckgabe

_retnl() hat keinen R.ckgabewert.

Anwendung

_retnl() kopiert einen numerischen Wert in den R.ckgabebereich von Clipper. Dieser Wert ist unter Clipper als R.ckgabewert der C- oder Assembler-Routine verf.gbar, sobald die Routine die Ablaufkontrolle zur.ckgegeben hat.

Beispiele

In C:

```
_retnl(123456789L);
```

In Assembler:

```
EXTRN   __retnl:FAR
  mov   dx, 1883
  mov   ax, 52501
  push  dx
  push  ax
  call  __retnl
  add   sp, 4          ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_storc() Speichert einen String in einer referenzierten Clipper-Variablen

---

C-Prototyp

```
#include "extend.h"
void _storc(char *string, int paramNum [, int arrayIndex])
```

Parameter

string ist ein far-Pointer auf eine Zeichenfolge, die in einer Clipper-Variablen vom Typ character gespeichert werden soll.

paramNum ist die Position des Parameters in der aktuellen Parameterliste.

arrayIndex ist ein Array-Index, um ein bestimmtes Element anzusprechen, falls ein Array .bergeben wurde.

R.ckgabe

_storc() gibt 1 zur.ck, wenn die Funktion erfolgreich war, ansonsten 0.

Anwendung

_storc() weist einen String einer Clipper-Variablen zu, die per Referenz .bergeben wurde. Ist der spezifizierte Parameter nicht per Referenz .bergeben worden, gibt _storc() 0 zur.ck.

_storc() durchsucht den String nach dem ersten NULL-Byte (ASCII 0), um so die logische L.nge der Zeichenkette festzustellen. Sollen bin.re Daten, die mehrere NULL-Bytes enthalten, zur.ckgegeben werden, mu. _storclen() benutzt werden.

Achtung

_storc() alloziert Speicher im Clipper-Heap und erzeugt eine Kopie des .bergebenen Parameters. Der String selbst mu. nicht weiter existieren, sobald _storc() beendet wurde.

Beispiele

In C:

```
_storc("Hallo Welt", 1);
```

In Assembler:

```
EXTRN   __storc:FAR
  mov   ax, 1
  push  ax              ; Parameternummer .bergeben
  mov   dx, seg MyString
  mov   ax, offset MyString
  push  dx
  push  ax
  call  __storc
  add   sp, 6           ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_storclen() Speichert einen String bestimmter L.nge in einer referenzierten Clipper-Variablen

---

C-Prototyp

```
#include "extend.h"
void _storclen(char *string, unsigned int length,
  int paramNum [, int arrayIndex])
```

Parameter

string ist ein far-Pointer auf eine Zeichenfolge, die in einer Clipper-Variablen vom Typ character gespeichert werden soll.

length gibt die L.nge der zu speichernden Zeichenfolge an.

paramNum ist die Position des Parameters in der aktuellen Parameterliste.

arrayIndex ist ein Array-Index, um ein bestimmtes Element anzusprechen, falls ein Array .bergeben wurde.

R.ckgabe

_storclen() gibt 1 zur.ck, wenn die Funktion erfolgreich war, ansonsten 0.

Anwendung

_storclen() weist einer Clipper-Variablen, die per Referenz .bergeben wurde, einen String zu. Ist der spezifizierte Parameter nicht per Referenz .bergeben worden, gibt _storclen() 0 zur.ck.

Da _storclen() die L.nge des Strings als explizites Argument erh.lt, mu. der String nicht durch ein NULL-Byte (ASCII 0) abgeschlossen sein. Aus diesem Grunde kann der String auch ein oder mehrere NULL-Bytes beinhalten.

Achtung

_storclen() alloziert Speicher im Clipper-Heap und erzeugt eine Kopie des .bergebenen Parameters. Der String selbst mu. nicht weiter existieren, sobald _storclen() beendet wurde.

Beispiele

In C:

```
char data[5] = {1, 0, 2, 0, 3, 3}
_storclen(data, 5, 1);
```

In Assembler:

```
EXTRN   __storclen:FAR
  mov   ax, 1
  push  ax              ; Parameternummer .bergeben
  mov   ax, 5
  push  ax
  mov   dx, seg MyData
  mov   ax, offset MyData
  push  dx
  push  ax
  call  __storclen
  add   sp, 8           ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_stords() Speichert einen String als Datumswert in einer referenzierten Clipper-Variablen

---

C-Prototyp

```
#include "extend.h"
void _stords(char *string, int paramNum [, int arrayIndex])
```

Parameter

string ist ein far-Pointer auf einen null-terminierten String, der den Datumswert in der Form JJJJMMTT repr.sentiert.

paramNum ist die Position des Parameters in der aktuellen Parameterliste.

arrayIndex ist ein Array-Index, um ein bestimmtes Element anzusprechen, falls ein Array .bergeben wurde.

R.ckgabe

_stords() gibt 1 zur.ck, wenn die Funktion erfolgreich war, ansonsten 0.

Anwendung

_stords() weist einer Clipper-Variablen, die per Referenz .bergeben wurde, einen String zu. Ist der spezifizierte Parameter nicht per Referenz .bergeben worden, gibt _stords() 0 zur.ck.

_stords() konvertiert den String in einen Clipper-Datumswert. Falls der String kein g.ltiges Datum darstellt, wird ein leerer Datumswert erzeugt.

Achtung

_stords() alloziert Speicher im Clipper-Heap und erzeugt eine Kopie des .bergebenen Parameters. Der String selbst mu. nicht weiter existieren, sobald _stords() beendet wurde.

Beispiele

In C:

```
_stords("19903112", 1);
```

In Assembler:

```
EXTRN   __stords:FAR
  mov   ax, 1
  push  ax              ; Parameternummer .bergeben
  mov   dx, seg MyDateString
  mov   ax, offset MyDateString
  push  dx
  push  ax
  call  __stords
  add   sp, 6           ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_storl() Speichert einen Integer als logischen Wert in einer referenzierten Clipper-Variablen

---

C-Prototyp

```
#include "extend.h"
void _storl(int logical, int paramNum [, int arrayIndex])
```

Parameter

logical ist ein Integer, der den logischen Wert darstellt. Eine Null wird als .F., jeder andere Wert als .T. .bersetzt.

paramNum ist die Position des Parameters in der aktuellen Parameterliste.

arrayIndex ist ein Array-Index, um ein bestimmtes Element anzusprechen, falls ein Array .bergeben wurde.

R.ckgabe

_storl() gibt 1 zur.ck, wenn die Funktion erfolgreich war, ansonsten 0.

Anwendung

_storl() weist einer Clipper-Variablen, die per Referenz .bergeben wurde, einen C-Integer als logischen Wert zu. Ist der spezifizierte Parameter nicht per Referenz .bergeben worden, gibt _storl() 0 zur.ck.

Beispiele

In C:

```
_storl(1, 1);
```

In Assembler:

```
EXTRN   __storl:FAR
  mov   ax, 1
  push  ax              ; Parameternummer .bergeben
  push  ax              ; logischen Wert .bergeben
  call  __storl
  add   sp, 4           ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_stornd() Speichert einen Double-Wert als numerischen Wert in einer referenzierten Clipper-Variablen

---

C-Prototyp

```
#include "extend.h"
void _stornd(double n, int paramNum [, int arrayIndex])
```

Parameter

n ist ein numerischer Ausdruck vom Typ Double.

paramNum ist die Position des Parameters in der aktuellen Parameterliste.

arrayIndex ist ein Array-Index, um ein bestimmtes Element anzusprechen, falls ein Array .bergeben wurde.

R.ckgabe

_stornd() gibt 1 zur.ck, wenn die Funktion erfolgreich war, ansonsten 0.

Anwendung

_stornd() weist einer Clipper-Variablen, die per Referenz .bergeben wurde, einen Double-Wert als numerischen Wert zu. Ist der spezifizierte Parameter nicht per Referenz .bergeben worden, gibt _stornd() 0 zur.ck.

Beispiele

In C:

```
_stornd( (double) 3.14, 1);
```

In Assembler:

```
EXTRN   __stornd:FAR
  mov   ax, 1
  push  ax              ; Parameternummer .bergeben
  push  word ptr (MyPi6)
  push  word ptr (MyPi4)
  push  word ptr (MyPi2)
  push  word ptr MyPi
  call  __stornd
  add   sp, 10          ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_storni() Speichert einen Integer als numerischen Wert in einer referenzierten Clipper-Variablen

---

C-Prototyp

```
#include "extend.h"
void _storni(int n, int paramNum [, int arrayIndex])
```

Parameter

n ist ein numerischer Ausdruck vom Typ Integer.

paramNum ist die Position des Parameters in der aktuellen Parameterliste.

arrayIndex ist ein Array-Index, um ein bestimmtes Element anzusprechen, falls ein Array .bergeben wurde.

R.ckgabe

_storni() gibt 1 zur.ck, wenn die Funktion erfolgreich war, ansonsten 0.

Anwendung

_storni() weist einer Clipper-Variablen, die per Referenz .bergeben wurde, einen Integer als numerischen Wert zu. Ist der spezifizierte Parameter nicht per Referenz .bergeben worden, gibt _storni() 0 zur.ck.

Beispiele

In C:

```
_storni( 123, 1);
```

In Assembler:

```
EXTRN   __storni:FAR
  mov   ax, 1
  push  ax              ; Parameternummer .bergeben
  mov   ax, 123
  push  ax
  call  __storni
  add   sp, 4           ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

ASC() Ermittelt den ASCII-Wert eines Zeichens

Syntax

ASC(<cZeichen>) -> nCode

Parameter

<cZeichen> ist das Zeichen, dessen ASCII-Wert ermittelt werden soll.

R.ckgabe

ASC() gibt einen ganzzahligen, numerischen Wert zwischen 0 und 255 zur.ck, der den ASCII-Wert von <cZeichen> repr.sentiert.

Anwendung

ASC() ist eine konvertierende Funktion, die den ASCII-Wert des ersten, linksb.ndigen Zeichens in einer Zeichenfolge liefert. ASC() ist die umgekehrte Funktion von CHR(). ASC() wird meistens dann benutzt, wenn Berechnungen mit ASCII-Werten von Zeichen oder Zeichenfolgen vorgenommen werden sollen.

Beispiele

Das Beispiel zeigt die verschiedenen Resultate der Funktion ASC():

```
? ASC("A")              // Resultat: 65
? ASC("Apfel")          // Resultat: 65
? ASC("a")              // Resultat: 97
? ASC("Z") - ASC("A")   // Resultat: 25
? ASC("")               // Resultat: 0
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

_stornl() Speichert einen Long Integer als numerischen Wert in einer referenzierten Clipper-Variablen

---

C-Prototyp

```
#include "extend.h"
void _stornl(long int n, int paramNum [, int arrayIndex])
```

Parameter

n ist ein numerischer Ausdruck vom Typ Long Integer.

paramNum ist die Position des Parameters in der aktuellen Parameterliste.

arrayIndex ist ein Array-Index, um ein bestimmtes Element anzusprechen, falls ein Array .bergeben wurde.

R.ckgabe

_stornl() gibt 1 zur.ck, wenn die Funktion erfolgreich war, ansonsten 0.

Anwendung

_stornl() weist einer Clipper-Variablen , die per Referenz .bergeben wurde, einen Long Integer als numerischen Wert zu. Ist der spezifizierte Parameter nicht per Referenz .bergeben worden, gibt _stornl() 0 zur.ck.

Beispiele

In C:

```
_stornl( 123456789L, 1);
```

In Assembler:

```
EXTRN   __stornl:FAR
  mov   ax, 1
  push  ax              ; Parameternummer .bergeben
  mov   dx, 1883
  mov   ax, 52501
  push  dx
  push  ax
  call  __stornl
  add   sp, 6           ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_xalloc() Alloziert Speicher und gibt NULL zur.ck, falls erfolglos

---

C-Prototyp

```
#include "extend.h"
void far *_xalloc(unsigned int size)
```

Parameter

size ist die zu allozierende Speichergr..e in Byte.

R.ckgabe

_xalloc() gibt einen Pointer auf den allozierten Speicher zur.ck.

Anwendung

_xalloc() alloziert einen Speicherbereich mit der Gr..e size im Clipper-Heap. Im Erfolgsfall wird ein Pointer auf den allozierten Speicher zur.ckgegeben, andernfalls NULL.

Um mit _xalloc() allozierten Speicher wieder freizugeben, benutzen Sie bitte _xfree().

Beispiele

In C:

```
char *mem;
mem = _xalloc(32);
```

In Assembler:

```
EXTRN   __xalloc:FAR
  mov   ax, 32
  push  ax
  call  __xalloc          ; Pointer in DX:AX
  add   sp, 2             ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_xfree() Freigabe von alloziertem Speicher

C-Prototyp

```
#include "extend.h"
void far *_xfree(void far *mem)
```

Parameter

mem ist ein far-Pointer auf einen mit _xalloc() oder _xgrab() allozierten Speicherbereich.

R.ckgabe

_xalloc() hat keinen R.ckgabewert.

Anwendung

_xalloc() gibt einen durch _xalloc() oder _xgrab() vorher allozierten Speicherbereich wieder an Clipper zur.ck. Der von diesen Funktionen erzeugte Pointer mu. an _xalloc() als Argument .bergeben werden.

Achtung

Um mit Clipper Sommer '87 kompatibel zu sein, ist in EXTEND.H _exmback() durch #define als _xfree() definiert.

Beispiele

In C:

```
char *mem;
mem = _xgrab(32);
_xfree(mem);
```

In Assembler:

```
EXTRN   __xgrab:FAR
EXTRN   __xfree:FAR
  mov   ax, 32
  push  ax
  call  __xgrab          ; Pointer in DX:AX
  add   sp, 2            ; Stack zur.cksetzen
  push  dx
  push  ax
  call  __xfree          ; Speicher zur.ckgeben
  add   sp, 4            ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

_xgrab() Alloziert Speicher und generiert einen Fehler, falls erfolglos

---

C-Prototyp

```
#include "extend.h"
void far *_xgrab(unsigned int size)
```

Parameter

size ist die zu allozierende Speichergr..e in Byte.

R.ckgabe

_xgrab() gibt einen Pointer auf den allozierten Speicher zur.ck.

Anwendung

_xgrab() alloziert einen Speicherbereich in size Bytes im Clipper-Heap. Im Erfolgsfall wird ein Pointer auf den allozierten Speicher zur.ckgegeben, andernfalls generiert _xgrab() einen Laufzeitfehler (es wird immer ein Pointer zur.ckgegeben und nicht NULL).

Um mit _xgrab() allozierten Speicher wieder freizugeben, benutzen Sie bitte _xfree().

Achtung

Um mit Clipper Sommer '87 kompatibel zu sein, ist in EXTEND.H _exmgrab() durch #define als _xgrab() definiert.

Beispiele

In C:

```
char *mem;
mem = _xgrab(32);
```

In Assembler:

```
EXTRN   __xgrab:FAR
  mov   ax, 32
  push  ax
  call  __xgrab          ; Pointer in DX:AX
  add   sp, 2            ; Stack zur.cksetzen
```

Dateien

Bibliothek ist CLIPPER.LIB, Header-Datei ist EXTEND.H

See Also:

#command | #translate
Spezifiziert einen anwenderdefinierten Befehl oder eine .bersetzungs-Direktive

---

Syntax
#command <AustauschMuster> => <ErgebnisMuster>
#translate <AustauschMuster> => <ErgebnisMuster>

Parameter
<AustauschMuster> ist das auszutauschende Muster im Quelltext und wird auch als match-pattern bezeichnet.

<ErgebnisMuster> ist das Muster eines Textes, der erzeugt wird, wenn ein Teil des Quelltextes mit dem Muster in <AustauschMuster> .bereinstimmt.

Das Symbol => zwischen <AustauschMuster> und <ErgebnisMuster> ist ein fester Bestandteil in der Syntax der #command- und #translate-Direktiven und mu. angegeben werden. Das Symbol besteht aus einem Gleichheitszeichen gefolgt von einem Gr..erzeichen; dazwischen d.rfen keine Leerzeichen angegeben sein. Dieses Symbol ist nicht mit den Symbolen >= und <= zu verwechseln, die unter Clipper als Vergleichs-Operatoren verwendet werden.

Anwendung
#command und #translate sind .bersetzungs-Direktiven. Sie werden unter Clipper benutzt, um Befehle und Pseudo-Funktionen zu definieren. Die Deklarationsregel enth.lt zwei Teile, ein Austauschmuster und ein Ergebnismuster. Das <AustauschMuster> wird verwendet, um im Quelltext, in der Regel eine Programmdatei, nach passenden Befehlen zu suchen und Teile des Befehlstextes zur Ermittlung des Ergebnistextes heranzuziehen.

Das <ErgebnisMuster> gibt an, was als Ergebnistext ermittelt werden soll und vor allem auch, wie mit Hilfe der gesichterten Teile ermittelt werden soll.

#command und #translate sind sich sehr .hnlich, unterscheiden sich jedoch in dem Verfahren, nach dem eine .bereinstimmung gepr.ft werden soll. Die #command-Direktive pr.ft eine .bereinstimmung nur dann, wenn der Quelltext eine komplette Anweisung enth.lt, #translate hingegen kann auch Teile einer Anweisung ersetzen. Damit eignet sich #command besonders zur Definition kompletter Befehle, w.hrend #translate zur Bestimmung von Vereinbarungen und Pseudo-Funktionen, die nicht in Form einer kompletten Anweisung auftreten, eingesetzt werden kann.

#command und #translate haben den gleichen G.ltigkeitsbereich wie eine #define-Direktive. Eine solche Definition gilt nur innerhalb einer Programmdatei (.PRG), es sei denn, sie wurde in STD.CH oder in einer anderen Header-Datei angegeben und das Programm mit dem Compilerschalter /U compiliert. Sind Direktiven nicht in einer Header-Datei definiert, gelten diese von der Zeile, in der sie deklariert wurden, bis zum Ende einer solchen Programmdatei. Eine Definition durch die Direktiven #command und #translate kann, im Gegensatz zu einem #define, nicht .ber ein #undef aufgehoben werden.

Austauschmuster (match-pattern)
Das <AustauschMuster> als Teil einer #command- oder #translate-Direktive spezifiert das Muster, nach dem .bereinstimmungen im Quelltext gefunden werden sollen. Im folgenden sind die m.glichen Bestandteile des Austauschmusters und deren .bersetzung beschrieben:

Literale Werte sind Zeichenfolgen, die in einem Austauschmuster als Text aufgef.hrt sind. In einer #translate-Direktive m.ssen die Zeichen mit dem Quelltext exakt .bereinstimmen.

Worte sind Schl.sselworte und Bezeichner, die entprechend der dBASE-Konvention (Gro.-/Kleinschreibung unabh.ngig, die ersten vier Zeichen signifikant)

verglichen werden. Das Austauschmuster mu. mit einem Wort beginnen.

Match-Marker sind Markierungen, die einem Teil des Quelltextes einen Namen zuordnen, wobei die Form vom Typ des Markers bestimmt wird. Der Name des Markers kann im Ergebnismuster zur Ersetzung des entgegengenommenen Textes verwendet werden, um so zum Beispiel Argumente an Funktionen zu .bergeben. Die Namen von Markern sind Bezeichner und m.ssen daher der f.r Clipper festgelegten Namenskonvention folgen. Kurz: die Namen m.ssen mit einem Buchstaben oder Unterstrich-Zeichen beginnen und nur die ersten zehn Stellen sind signifikant.

Alle Typen von Match-Markern sind nachfolgend aufgef.hrt. Zur Identifizierung m.ssen sie in spitzen Klammern(<>) stehen.

Tabelle 3-1: Match-Marker

Match-Marker	Bezeichnung
<idMarker>	Regular Match-Marker
<idMarker, ...>	List Match-Marker
<idMarker:WortListe>	Restricted Match-Marker
<*idMarker*>	Wild Match-Marker
<(idMarker)>	Extended Expression Match-Marker

. **Regular-Match-Marker:** Dieser Marker nimmt den n.chsten legalen Ausdruck im Quelltext entgegen. Der Regular Match-Marker ist der Marker mit dem meisten M.glichkeiten, weshalb er am h.ufigsten in einer Befehls-Deklaration verwendet wird.

. **List-Match-Marker:** Dieser Marker nimmt einen, durch Kommata getrennten, aufgelisteten und legalen Ausdruck (Liste) entgegen. Ist zu einem List-Match-Marker kein passender Eingabe-Text vorhanden, bleibt dieser Marker leer. Der List-Match-Marker wird zur Definition von Befehlsvereinbarungen verwendet, die eine Reihe von Argumenten enthalten.

. **Restricted-Match-Marker:** Dieser Marker nimmt eines der in einer Liste durch Kommata getrennt angegebenen Worte entgegen. Stimmt der Quelltext nicht mit mindestens einem der Worte .berein, ist der Inhalt des durch diesen Namen angegebenen Markers leer.

. **Wild-Match-Marker:** Dieser Marker nimmt einen beliebigen Quelltext ab der aktuellen Position bis zum Ende der Zeile entgegen. Wild-Match-Marker werden grunds.tzlich dazu benutzt, um Quelltexte entgegenzunehmen, die keinen legalen Ausdruck enthalten.

. **Extended Expression Match-Marker:** Dieser Marker nimmt einen regul.ren oder eine erweiterten Ausdruck entgegen, einschlie.lich Dateinamen oder Pfad-angaben. Er wird in Verbindung mit normalen Stringify-Result-Marker verwendet, um erweiterte Ausdr.cke in eine Zeichenkette entgegenzunehmen, oder wenn Zeichenkettenbegrenzer nicht explizit angegeben sind.

Optional-Match-Clauses sind Teile des Austausch-Musters, eingeschlossen in eckigen Klammern ([]). Solche Klauseln werden ben.tigt, um optionale Angaben innerhalb einer Befehlsdeklaration zu erm.glichen. Eine optionale Match-Klausel kann beliebige Teile des Austauschmusters einschlie.lich weiterer optionaler Klauseln umfassen. Es existieren zwei Arten von optionalen Match-Klauseln: eine wird definiert, beginnend mit einem Schl.sselwort gefolgt von einem Match-Marker; eine andere wird nur durch das Schl.sselwort selbst definiert.

Ergebnis-Muster

Das <ErgebnisMuster> als Teil einer .bersetzungs-Direktive definiert den Text, den der Pre-Prozessor erzeugen soll, wenn eine .bereinstimmung des Quelltextes mit dem <AustauschMuster> besteht.

<ErgebnisMuster> wird durch eine oder mehrere der folgenden Komponenten angegeben:

Literale Token sind Zeichenfolgen, die direkt in den Ergebnistext geschrieben werden.

Worte sind Schl.sselworte und Bezeichner, die direkt in den Ergebnistext geschrieben werden.

Result-Marker beziehen sich unmittelbar auf den Namen eines Match-Markers. Ein Teil des Quelltextes, der durch einen Match-Marker entgegengenommen wurde, ist .ber einen Result-Marker in den Ergebnistext zu schreiben. Es existieren mehrere Arten von Result-Markern, die im folgenden aufgelistet sind:

Tabelle 3-2: Result-Marker

Result-Marker	Bezeichnung
<idMarker>	Regular-Result-Marker
#<idMarker>	Dumb Stringify-Result-Marker
<"idMarker">	Normal Stringify-Result-Marker
<(idMarker)>	Smart Stringify-Result-Marker
<{idMarker}>	Blockify-Result-Marker
<.idMarker.>	Logify-Result-Marker

. **Regular-Result-Marker:** Dieser Marker schreibt einen entgegengenommenen Quelltext in den Zieltext. Falls kein passender Quelltext gefunden wurde, wird auch kein Zieltext generiert. Dies ist der gebr.uchlichste Result-Marker. Er wird verwendet, wenn keine besonderen Anforderungen bestehen. Der Regular-Result-Marker kann in Verbindung mit allen Match-Markern angewendet werden, wird jedoch vorzugsweise in Kombination mit einem Regular-Match-Marker eingesetzt.

. **Dumb Stringify-Result-Marker:** Der entgegengenommene Quelltext wird als Zeichenkettenausdruck in den Zieltext gestellt. Wurde kein Quelltext aufgenommen, wird eine leere Zeichenkette ("") in den Zieltext geschrieben. Wurde der Quelltext mit einem List-Match-Marker entgegengenommen, wird die komplette Liste als Liste von Zeichenketten in den Zieltext geschrieben.

. **Normal Stringify-Result-Marker:** Der entgegengenommene Quelltext wird als Zeichenkettenausdruck in den Zieltext gestellt. Wurde kein Quelltext entgegengenommen, wird auch nichts in den Zieltext geschrieben. Wurde der Quelltext mit einem List-Match-Marker entgegengenommen, wird jedes Element der Liste als eigene Zeichenkette in den Zieltext geschrieben.

. **Smart Stringify-Result-Marker:** Dieser Result-Marker schreibt einen entgegengenommenen Quelltext nur dann als Zeichenkette in den Zieltext, wenn der Quelltext in runden Klammern () steht. Wurde kein Quelltext aufgenommen, wird auch nichts in den Zieltext geschrieben. Wurde der Quelltext mit einem List-Match-Marker entgegengenommen, wird jedes Element der Liste als eigene Zeichenkette in den Zieltext geschrieben.

. **Blockify-Result-Marker:** Dieser Result-Marker schreibt einen entgegengenommenen Quelltext als Code-Block, ohne irgendwelche Argumente in den Zieltext. Der Text x + 3 w.rde also das Ergebnis {|| x + 3} liefern. Wurde kein Quelltext aufgenommen, wird auch nichts in den Zieltext geschrieben. Wurde der Quelltext mit einem List-Match-Marker entgegengenommen, wird jedes Element der Liste als eigener Code-Block in den Zieltext geschrieben.

. **Logify-Result-Marker:** Dieser Result-Marker schreibt ein true (.T.) in den Zieltext, wenn eine .bereinstimmung im Quelltext gefunden wurde, ansonsten wird false (.F.) in den Zieltext geschrieben. Der Quelltext selbst wird nicht in den Zieltext .bernommen.

Repeating-Result-Clauses sind Teile vom <ErgebnisMuster>, eingeschlossen in eckige Klammern. Der Text innerhalb der Wiederholungs-Klausel wird so

oft in den Ergebnistext geschrieben, wie Quelltext f.r einige oder alle Result-Marker in der Vereinbarung vorhanden ist. Ist kein entgegengenommener Text vorhanden, schreibt die Wiederholungsvereinbarung nichts in den Zieltext. Wiederholungs-Klauseln d.rfen nicht verschachtelt werden.

.ber Wiederholungs-Klauseln k.nnen innerhalb einer #command-Deklaration unterschiedlich viele Argumente in den Zieltext .bernommen werden. Der Quelltext kann mit beliebigen Match-Markern, mit Ausnahme der Restricted-Match-Marker, entgegengenommen und mit beliebigen, korrespondierenden Result-Markern in den Zieltext geschrieben werden. Ein typisches Beispiel ist die Definition der Befehle STORE und REPLACE in der Header-Datei STD.CH.

Hinweise

. **Spezielle Zeichen im Zieltext:** Werden die Zeichen (<) oder ([) in einem Ausdruck innerhalb von <ErgebnisMuster> ben.tigt, ist diesen ein Backslash (\) voranzustellen.

. **Mehrfach-Befehlszeilen:** Es d.rfen mehrere Anweisungen in einem <ErgebnisMuster> angegeben werden, wenn eine Trennung durch Semikolon erfolgt. Sind zwei aufeinanderfolgende Anweisungen in getrennten Zeilen angegeben, mu. die Anweisung in der ersten Zeile mit zwei aufeinanderfolgenden Semikolons abgeschlossen sein.

Beispiele

Die folgenden Beispiele demonstrieren eine Reihe von Grundtechniken, die zur Befehls-Deklaration mit der #command- oder #translate-Direktive verwendet werden.

. Das folgende Beispiel zeigt eine h.ufig vorkommende Art der Befehlsdefinition, bestehend aus Regular-Match-Markern in Verbindung mit Regular-Result-Markern:

```
  #command @ <oben>, <links>, <unten>, <rechts> BOX <BoxZeichen>;
  =>;
  CmdBox(<oben>, <links>, <unten>, <rechts>, <BoxZeichen> )
```

. Dieses Beispiel benutzt einen List-Match-Marker in Verbindung mit einem Regular-Result-Marker, um den ?-Befehl zu definieren:

```
  #command ? [<Liste,...>]  => QOUT( <Liste> )
```

. Dieses Beispiel zeigt, wie ein Restricted-Match-Marker in Verbindung mit einem Logify-Result-Marker verwendet werden kann, um eine optionale Vereinbarung f.r einen Befehl zu definieren. Wird f.r dieses Beispiel der Zusatz ADDITIVE angegeben, schreibt der Logify-Result-Marker true (.T.), ansonsten false (.F.) in den Zieltext:

```
  #command RESTORE FROM <MemDatei> [<opt:ADDITIVE>];
  =>;
  CmdRestore( <(MemDatei)>, <.opt.> )
```

. Dieses Beispiel zeigt, wie ein List-Match-Marker in Kombination mit einen Smart-Stringify-Result-Marker verwendet wird, um eine .ber FIELDS angegebene Feldliste als Argument einer Funktion in den Zieltext zu schreiben. In diesem Beispiel ist das Argument als Array definiert:

```
  #command COPY TO <Datei> [FIELDS <Felder,...>;
  =>;
  CmdCopyAll( <(Datei)>, { <(Felder)> } )
```

. Das n.chste Beispiel zeigt die Verwendung von Wild-Match-Markern, um Befehle ohne Wirkung zu deklarieren. Dies kann verwendet werden, wenn mit Clipper Quellprogramme compiliert werden sollen, die f.r andere Dialekte geschrieben wurden:

```
  #command SET ECHO <*Text*>  =>
  #command SET TALK <*Text*>  =>
```

. In diesen Beispielen wird die Verwendung von Wild-Match-Markern in Verbindung mit Dumb-Stringify-Result-Markern gezeigt, um literal angegebene Befehlsargumente entgegenzunehmen und im Zieltext als Zeichenkette abzulegen:

```
  #command SET PATH TO <*Pfad*> => SET( _SET_PATH, #<Pfad> )
  #command SET COLOR TO <*Farben*> => SETCOLOR( #<Farben> )
```

. Diese beiden Beispiele verwenden normale Stringify-Result-Marker in Verbindung mit Blockify-Result-Markern. Beide compilieren einen Ausdruck und sichern den Text des Ausdrucks f.r eine sp.tere Verwendung (z.B. Index Datei-Header):

```
  #command SET FILTER TO <exp>;
  =>;
  CmdSetFilter( <{exp}>, <"exp"> )

  #command INDEX ON <Schl.ssel> TO <IndexDatei>;
  =>;
  CmdCreateIndex(<(IndexDatei)>, <"Schl.ssel">, <{Schl.ssel}> )
```

. Dieses Beispiel zeigt die Verwendung eines Smart-Stringify-Result-Markers, um den USE-Befehl mit einem Argument zu implementieren, der als erweiterter Ausdruck angegeben werden kann:

```
  #command USE <db> [ALIAS <a>];
  =>;
  CmdOpenDbf( <(db)>, <(a)> )
```

. Dieses Beispiel zeigt, wie wichtig ein Blockify-Result-Marker f.r die Definition von Datenbankbefehlen ist. Hier wird er jeweils zur Erf.llung der FOR- und WHILE-Bedingungen angegeben:

```
  #command COUNT [TO <Var>];
    [FOR <for>] [WHILE <while>];
    [NEXT <next>] [RECORD <rec>] [<rest:REST>] [ALL];
  =>;
    <var> := 0;;
    DBEVAL( {|| <var++> }, <{for}>, <{while}>,;
                  <next>, <rec>, <.rest.> )
```

. Das Beispiel zeigt nochmals den USE-Befehl, um die optionalen Klauseln mit Schl.sselw.rtern im <AustauschMuster> zu demonstrieren. Die erste Vereinbarung enth.lt ein Schl.sselwort gefolgt von einem Argument, die zweite nur ein Schl.sselwort:

```
  #command USE <db> [<new:NEW>] [ALIAS <a>] [INDEX <index,...>];
    [<ex:EXCLUSIVE>] [<sh:SHARED>] [<ro:READONLY>];
  =>;
    CmdOpenDbf( <(db)>, <(a)>, <.new.>, ;
    IF( <.sh.> .OR. <.ex.>, !<.ex.>, NIL), <.ro.>, {<(index)>} )
```

. Dieses Beispiel zeigt die Beziehung zwischen einer optionalen Match-Klausel und einer Repeating-Result-Clause bei der Definition des STORE-Befehls:

```
  #command STORE <Wert> TO <var1> [, <varN> ];
  =>;
  <var1> := [ <varN> := ] <Wert>
```

. Dieses Beispiel zeigt, wie mit der #translate-Direktive eine Pseudofunktion erstellt werden kann:

```
  #translate ALLTRIM(<string>) => LTRIM(RTRIM( <string> ))
```

See Also:

ASCAN() Durchsucht ein Array nach einem Wert oder solange, bis ein Code-Block .T. liefert

---

Syntax

ASCAN(<aArray>, <expSuch> ,[<nStart>] ,[<nAnzahl>]) -> nPosition

Parameter

<aArray> ist das Array, das durchsucht werden soll.

<expSuch> ist entweder ein Wert, nach dem gesucht werden soll, oder ein Codeblock. Wird mit <expSuch> ein Wert angegeben, so kann dieser vom Typ Zeichen, Datum, Logisch oder Numerisch sein.

<nStart> ist das Element, bei dem die Suche beginnen soll. Wird dieser Parameter nicht angegeben, beginnt die Suche bei dem ersten Element.

<nAnzahl> ist die Anzahl der Elemente, beginnend bei <nStart>, die durchsucht werden sollen. Wird dieser Parameter nicht angegeben, so wird bis zum Ende des Arrays durchsucht.

R.ckgabe

ASCAN() gibt einen numerischen Wert zur.ck, welcher der Position des zuletzt durchsuchten Elements entspricht. Wird als <expSuch> ein einfacher Wert angegeben, liefert ASCAN() die Position des zuerst gefundenen Wertes oder 0, wenn nichts gefunden wurde. Ist <expSuch> ein Codeblock, liefert ASCAN() die Position des Elements, bei dem der Codeblock true (.T.) ergibt.

Anwendung

ASCAN() durchsucht ein Array nach einem bestimmten Wert. Die Art des Suchvorganges ist dabei .hnlich der von FIND oder SEEK. <expSuch> wird mit dem Element eines Arrays von links beginnend verglichen, bis keine Zeichen mehr .brig sind. Wird keine .bereinstimmung gefunden, f.hrt ASCAN() mit dem n.chsten Element fort.

Da ASCAN() beim Vergleich den Operator = benutzt, ist der Erfolg der Suche abh.ngig von der Einstellung von EXACT. Ist der Status EXACT ON, m.ssen die beiden Ausdr.cke Zeichen f.r Zeichen .bereinstimmen.

Wenn mit <expSuch> ein Codeblock als Parameter .bergeben wird, wird f.r jedes Element, auf das zugegriffen wird, der Codeblock ausgef.hrt, wobei der Wert des jeweiligen Elements dem Codeblock .bergeben wird. Das Durchsuchen des Arrays wird beendet, wenn der Codeblock true (.T.) liefert, oder nachdem das letzte Element durchsucht wurde.

Beispiele

. Das Beispiel erzeugt zun.chst ein Array mit drei Elementen und zeigt das Durchsuchen dieses Arrays mittels einfachen Werten und einem Codeblock:

```
aArray  := {"Thomas", "Peter", "Anne"}
? ASCAN(aArray, "Peter")            // Resultat 2
? ASCAN(aArray, "peter")            // Resultat 0
bBlock  := {  | x UPPER(x) == "PETER" }
? ASCAN(aArray, bBlock)             // Resultat 2
```

. Ein weiteres Beispiel demonstriert, wie mit dem Durchsuchen fortgefahren werden kann, wenn nach dem mehrfachen Auftreten eines Begriffes gesucht wird:

```
LOCAL aArray := {"Thomas", "Peter", "Anne", "Peter"}, nStart := 1
nEnd := LEN(aArray)
DO WHILE (nPos := ASCAN(aArray, "Peter", nStart)) > 0
  ? nPos, aArray[nPos]
  IF (nStart := ++nPos) > nEnd
    EXIT
  ENDIF
```

```
    ENDDO
```

Dateien

```
  Bibliothek ist EXTEND.LIB
```

See Also:

#define Definiert eine symbolische Konstante oder eine Pseudo-Funktion

---

Syntax

#define <idKonstante> [<ErsatzText>]
#define <idFunktion>([<Argumente Liste>]) [<exp>]

Parameter

<idKonstante> ist der Name einer zu definierenden symbolischen Konstante.
<ErsatzText> ist der optionale Wert, gegen den die symbolische Konstante sp.ter ausgetauscht wird.

<idFunktion> ist die Definition einer Pseudofunktion mit einer optionalen <Argumente Liste>. Der angegebenen <idFunktion> mu. direkt (ohne Leerzeichen) eine offene runde Klammer, dann die <Argumente Liste> und eine geschlossene runde Klammer folgen.

<exp> ist der zu ersetzende Ausdruck. Dieser Ausdruck sollte in runde Klammern gestellt werden, um die korrekte Evaluierung des Ausdrucks zu garantieren.

Hinweis

#define-Bezeichner sind im Gegensatz zu #command und #translate case-sensitive, es wird also zwischen Gro.- und Kleinschreibung unterschieden.

Anwendung

Die #define-Direktive definiert einen Bezeichner und weist ihm optional eine Zeichenfolge zu, die ihn zur Compilierzeit ersetzen soll. Ist ein Ersatztext angegeben, wird der Bezeichner, .hnlich wie bei einem Texteditor, im Ergebnis-Text ausgetauscht. Jede Quellzeile der Programmdatei wird vom Pre-Prozessor bearbeitet und nach dem Vorkommen solcher Bezeichner durchsucht. Jeder gefundene Bezeichner wird ausgetauscht.

Namensbezeichner einer #define-Direktive m.ssen den Namenskonventionen von Clipper folgen. Sie k.nnen aus Kombinationen von alphabetischen- und numerischen Zeichen sowie dem Unterstrich gebildet werden. Wie auch bei anderen Bezeichnern sind unter Clipper nur die ersten zehn Zeichen signifikant, es werden also nur die ersten zehn Zeichen unterschieden. Im Gegensatz zu anderen Clipper-Bezeichnern wird bei Bezeichnern der #define-Direktive zwischen Gro.- und Kleinschreibung unterschieden. Als empfohlene Konvention sollten solche Bezeichner nur in Gro.buchstaben angegeben werden, um Konflikte mit anderen Bezeichnern im Programm zu vermeiden. Des weiteren sollten sie mit einem oder zwei gleichen Zeichen beginnen, um sie so gruppieren und damit eindeutig identifizieren zu k.nnen. Beispiele k.nnen Sie jeder der zu Clipper 5.0 geh.renden Header-Dateien entnehmen, die Sie im Verzeichis CLIPPER5\INCLUDE finden. Jede #define-Definition mu. f.r sich in einer Zeile erfolgen, sie d.rfen nicht wie bei Anweisungen mehrfach in einer Zeile angegeben werden, indem sie durch ein Semikolon voneinander getrennt werden. Jeder #define-Direktive folgen ein oder mehrere Leerzeichen (oder auch TABs), ein eindeutiger Bezeichner und optional ein Ersatztext. Einmal definierte Bezeichner d.rfen in der Spezifikation anderer Bezeichner verwendet werden.

Ein #define-Bezeichner hat den gleichen G.ltigkeitsbereich wie eine STATIC-Variable mit Datei-weiter G.ltigkeit. Sie gilt nur f.r die jeweilige Programmdatei, in der sie definiert wurde, es sei denn, sie wird in der Header-Datei STD.CH oder in einer anderen, .ber die Compiler-Option /U angegebenen Header-Datei definiert. Im Gegensatz zu der dateiweit g.ltigen STATIC-Variablen ist ein #define-Bezeichner ab der Zeile sichtbar, in der er angegeben wird. Au.erdem ist er so lange g.ltig, bis er entweder mit #undef als ung.ltig erkl.rt, oder mit einer neuen Definition umdefiniert, bzw. durch das Ende der Programmdatei automatisch ung.ltig wird.

Existierende Bezeichner k.nnen umdefiniert oder als ung.ltig erkl.rt

werden. Eine Umdefinition erfolgt durch eine neue #define-Direktive, indem der Bezeichner einen neuen Ersatztext erh.lt. Die aktuelle Definition wird dadurch mit der neuen Definition .berschrieben und der Compiler gibt eine Warnung aus. So werden gegebenenfalls ungewollte Umdefinitionen auf jeden Fall sp.testens beim Compilieren bemerkt. #define-Direktiven dienen den folgenden Zwecken:

. Definition eines Kontroll-Bezeichners f.r #ifdef und #ifndef
. Definition von symbolischen Konstanten - ein Bezeichner steht f.r einen konstanten Wert
. Definition einer Pseudo-Funktion

Pre-Prozessor Kontroll-Bezeichner
Die einfachste Definition einer #define-Direktive ist die Definition ohne <ErsatzText>. Dieser Typ eines Bezeichners kann verwendet werden, wenn Bedingungen f.r #ifdef oder #ifndef geschaffen werden sollen. So k.nnen bestimmte Programmteile in Abh.ngigkeit von einem solchen Bezeichner compiliert oder aber nicht compiliert werden. Diese Art von Bezeichner wird oft auch in Verbindung mit dem Kommandozeilen-Schalter /D benutzt (siehe auch die Beispiele).

Symbolische Konstanten
Diese Art der #define-Direktive ist eine Namenszuweisung f.r konstante Werte. Diese Variante eines Bezeichners ist die Festlegung von symbolischen Konstanten. Ein Beispiel f.r die Symbolisierung von Konstanten ist die Bezeichnung von INKEY()-Tastencodes (siehe auch Datei INKEY.CH):

```
#define K_ESC 27

IF LASTKEY() = K_ESC
__.
  .<Anweisungen>...
__.
ENDIF
```

Wird dieser Bezeichner vom Pre-Prozessor im Quellprogramm aufgefunden, so ersetzt er diesen Bezeichner durch den konstanten Wert.

Obwohl dies auch durch die Definition von Variablen erreicht werden kann, hat die Verwendung von Konstanten-Definitionen einige Vorteile. Zum ersten kann der Compiler wesentlich schnelleren und kompakteren Code erzeugen als bei der Verwendung von Variablen. Zum zweiten ben.tigt die Verwendung von Variablen kostbaren Speicherplatz, was bei der Verwendung von symbolischen Konstanten nicht der Fall ist. Dar.ber hinaus ist die Verwendung von Variablen konzeptionell inkonsequent. Der Sinn einer Variablen ist, da. ihr Wert sich zur Laufzeit .ndern kann, was bei symbolischen Konstanten nicht der Fall ist.

Die Verwendung von symbolischen Konstanten anstelle normaler Konstanten hat einige Vorteile. Erstens erh.ht dies die Lesbarkeit der Programme. Im obigen Beispiel ist die Angabe .ber eine symbolische Konstante besser lesbar, als wenn ein numerischer INKEY()-Code angegeben w.rde. Zweitens erfolgt die Definition einer symbolischen Konstanten an zentraler Stelle, was die Pflege, Wartung und Portierbarkeit von Programmen wesentlich erh.ht. Der dritte Gund ist, da. sich mit symbolischen Konstanten Implementierungen und Umgebungsspezifikationen isolieren lassen.

Compiler Pseudo-Funktionen
Mit der #define-Direktive k.nnen nicht nur Substitutionen konstanter Werte vorgenommen werden, sie kann auch zur Definition von Pseudo-Funktionen dienen, die zur Compilierzeit aufgel.st werden. Die Definition einer Pseudo-Funktion erfolgt durch die Angabe des Bezeichners, unmittelbar gefolgt von einer offenen runden Klammer (ohne zwischengestellte Leerzeichen), einer Argumentenliste, einer runden geschlossenen Klammer und einem einzusetzenden Ausdruck. Zum Beispiel:

```
#define FLAECHE(nL.nge * nBreite) (nL.nge * nBreite)
#define SETVAR( x, y )  (x := y)
#define MAX( x, y ) (IF( x > y, x, y ))
```

Pseudo-Funktionen unterscheiden sich von symbolischen Konstanten durch die angegebenen Argumente. Wann immer der Pre-Prozessor einen Funktionsaufruf findet, der mit der Definition einer Pseudo-Funktion .bereinstimmt, ersetzt er diesen Funktionsaufruf durch den definierten Ersatz-Ausdruck. Die Argumente der Funktion werden .ber die angegebenen Namen in den Ersatz-Ausdruck eingebracht. Wenn der Ersatz-Ausdruck durch eine Pseudo-Funktion ersetzt wird, werden .bereinstimmende Namen durch den Textinhalt der Argumente ersetzt. Die folgende Funktionsaufrufe:

```
? FLAECHE(10, 12)
SETVAR(nWert, 10)
? MAX( 10, 9 )
```

werden ersetzt durch:

```
? (10 * 12)
(nWert := 10)
? (IF( 10 > 9, 10, 9))
```

Wenn eine Pseudo-Funktion mit #define definiert wird, sind einige Punkte zu beachten. Erstens ist es wichtig, da. man den Ergebnis-Ausdruck in runde Klammern einschlie.t, damit in einem Ausdruck das Ergebnis in der richtigen Reihenfolge evaluiert wird. Dies ist speziell bei numerischen Ausdr.cken sehr wichtig. Zweitens m.ssen in einer Pseudo-Funktion alle Argumente angegeben sein. Wird ein Argument nicht angegeben, kann der Pre-Prozessor diese Funktion nicht finden und eine Ersetzung findet nicht statt. Dies kann als eine der Unzul.nglichkeiten von Pseudofunktionen betrachtet werden.

Soll entschieden werden, ob besser eine Pseudo-Funktion oder eine anwenderdefinierte Funktion verwendet wird, sind einige Punkte zu ber.cksichtigen. Pseudo-Funktionen ben.tigen nicht den Overhead eines normalen Funktionsaufrufes und sind daher nicht nur generell schneller, sondern ben.tigen auch weniger Speicherplatz. Allerdings gestaltet sich eine eventuelle Fehlersuche mit dem Clipper-Debugger als wesentlich schwieriger. Pseudo-Funktionen haben im Vergleich zu anwenderdefinierten Funktionen einen anderen G.ltigkeitsbereich, sie erlauben kein Auslassen von Parametern und sind abh.ngig von der Gro.-/Kleinschreibung.

Einige dieser Unzul.nglichkeiten k.nnen durch Definition mit der #translate-Direktive ausgeglichen werden. Zum einen sind Pseudo-Funktionen, die .ber #translate definiert werden, nicht abh.ngig von der Gro.-/Kleinschreibung, zum anderen erlauben sie optionale Parameter und gehorchen der bekannten Vierzeichen-Regel (Abk.rzungen).

N.heres hier.ber erfahren Sie in der Beschreibung der #translate-Direktive in diesem Kapitel.

Beispiele

. Das folgende Beispiel zeigt, wie die Existenz einer symbolischen Konstante dazu benutzt werden kann, um eine bedingte Compilation durchzuf.hren:

```
  #define DEBUG
  .
  .<Anweisungen>...
  .
  #ifdef DEBUG
    Bezug(FILE("System.dbf"))
  #endif
```

. Dieses Beispiel zeigt, wie eine symbolische Konstante definiert und sp.ter durch den INKEY()-Code ersetzt wird:

```
  #define K_ESC 27
  .
  .<Anweisungen>...
  .
  IF INKEY() <> K_ESC
     DoIt()
  ELSE
     StopIt()
  ENDIF
```

. Dieses Beispiel demonstriert die Definition einer Pseudo-Funktion f.r die Standard Clipper-Funktionen MAX() und ALLTRIM():

```
  #define MAX(arg1, arg2) (IF(arg1 > arg2, arg1, arg2))
  #define ALLTRIM(cString)  (RTRIM(LTRIM(cString)))
  .
  .<Anweisungen>...
  .
  ? MAX(1, 2)
  ? ALLTRIM("   Hallo   ")
```

See Also:

#ifdef Compiliert einen Programm-Abschnitt, wenn ein Bezeichner definiert wurde

---

Syntax

```
#ifdef <Bezeichner>
  <Anweisungen>...
[#else]
  <Anweisungen>...
[#endif]
```

Parameter

<Bezeichner> ist der Name eines Bezeichners, dessen Existenz gepr.ft wird.

Anwendung

#ifdef...#endif erm.glicht das, was oft als bedingte Compilation oder conditional compilation bezeichnet wird. Dabei wird ein bestimmter Abschnitt des Quellprogrammes nur dann compiliert, wenn der angegebene <Bezeichner> zuvor definiert wurde. Ein solcher Bezeichner kann .ber die #define-Direktive, aber auch bei Aufruf des Compilers .ber die Compiler-Option /D innerhalb der Kommandozeile definiert werden.
Die #else-Direktive leitet einen Programmabschnitt ein, der compiliert wird, falls <Bezeichner> nicht definiert wurde. #endif gibt das Ende des bedingt zu compilierenden Programmabschnittes an.
Bedingte Compilation ist sehr empfehlenswert, wenn verschiedene Versionen eines Programmes erstellt werden sollen. So kann zum Beispiel der Programm-Code f.r eine Demo-Version und eine Voll-Version im gleichen Quellprogramm angegeben werden, wenn der jeweils unterschiedliche Programmcode in ein Konstrukt aus #ifdef-/#else-/#endif-Direktiven gestellt wird. Durch eine einzige Definition eines Bezeichners lassen sich dann unterschiedliche Programmversionen erstellen.

Beispiele

. Dieser Programmteil demonstriert den grunds.tzlichen Aufbau einer bedingten Compilation mit der #ifdef-Direktive:

```
  #define DEMO
  .
  .<Anweisungen>...
  .
  #ifdef DEMO
    .
    .<Anweisungen nur f.r das Demo-Programm>
    .
  #endif
```

. Dieses Beispiel zeigt, wie ein Bezeichner in der Kommandozeile des Compilers .ber die Compiler-Option /D definiert wird:

Unter DOS:

```
  C:>CLIPPER MeinProg /DDEBUG
```

Im Quellprogramm:

```
  #ifdef DEBUG
    Bezug(FILE("System.dbf"))
  #endif
```

. Dieses Beispiel zeigt, wie mit Hilfe von #ifdef einer symbolischen Konstante ein bestimmter Wert zugewiesen wird, falls ein Bezeichner definiert ist, und ein anderer Wert, wenn dies nicht der Fall ist:

```
  #ifdef M_MARGIN
    #undef                                M_MARGIN
    #define                                                                    _MARGIN  15
```

```
  #else
    #define                                                                    _MARGIN  10
  #endif
```

See Also:

#ifndef Compiliert einen Programm-Abschnitt, wenn ein Bezeichner nicht definiert wurde

---

Syntax

```
#ifndef <Bezeichner>
  <Anweisungen>...
[#else]
  <Anweisungen>...
[#endif]
```

Parameter

<Bezeichner> ist der Name des Bezeichners, dessen Existenz gepr.ft wird.

Anwendung

#ifndef...#endif erm.glicht eine bedingte Compilation. Es wird ein bestimmter Abschnitt des Quellprogrammes nur dann compiliert, wenn der angegebene <Bezeichner> nicht definiert wurde. Ein solcher Bezeichner kann .ber die #define-Direktive, oder bei Aufruf des Compilers durch die Compiler-Option /D innerhalb der Kommandozeile definiert werden.

Die #else-Direktive leitet einen Programmabschnitt ein, der compiliert wird, falls <Bezeichner> definiert wurde. #endif gibt das Ende des bedingt zu compilierenden Programmabschnittes an.

Die Direktive #ifndef arbeitet genau umgekehrt wie die Direktive #ifdef.

Beispiele

. Dieser Programmteil demonstriert den grunds.tzlichen Aufbau einer bedingten Compilation durch die #ifndef-Direktive:

```
  #define DEMO
  .
  .<Anweisungen>...
  .
  #ifndef DEMO
  ____.
     .<Anweisungen nur f.r die Vollversion des Programmes>
  ____.
  #else
  ____.
     .<Anweisungen nur f.r das Demo-Programm>
  ____.
  #endif
```

. Das Beispiel zeigt, wie ein Bezeichner in der Kommandozeile des Compilers .ber die Compiler-Option /D definiert wird:

Unter DOS:

```
  C>CLIPPER MeinProg /DNODEBUG
```

Im Quellprogramm:

```
  #ifndef NODEBUG
     Bezug(FILE("System.dbf"))
  #endif
```

. Das Beispiel zeigt, wie mit #ifndef eine symbolische Konstante in Verbindung mit der Kommandozeilenoption /D umdefiniert werden kann:

```
  #ifndef M_MARGIN
     #define M_MARGIN  15
  #endif
```

See Also:

#include Bezieht eine Datei mit in die aktuelle Programmdatei ein

---

Syntax

#include "<HeaderDateiAngabe>"

Parameter

<HeaderDateiAngabe> Gibt den Namen einer weiteren Quelldatei an, die in die aktuelle Quelldatei mit einbezogen werden soll. Dabei mu. der Dateiname in Anf.hrungszeichen gestellt werden. .hnlich wie bei anderen Dateispezifikationen kann der Name der Header-Datei einen Laufwerks-bezeichner, einen Verzeichnisnamen und eine Dateinamenserweiterung enthalten. Wenn nicht explizit ein Verzeichnis angegeben ist, sucht der Pre-Prozessor diese Datei an folgenden Stellen:

. Quelldatei-Verzeichnis
. Durch die Option /I angegebenes Verzeichnis
. Durch die DOS-Umgebungsvariable INCLUDE angegebenes Verzeichnis

#include-Direktiven k.nnen bis zu einer Tiefe von 16 Ebenen miteinander verschachtelt werden. Das bedeutet, da. eine .ber #include einbezogene Datei wiederum eine #include-Direktive enthalten darf usw., bis zu 16 Datei-Ebenen.

Anwendung

Die #include-Direktive f.gt den Inhalt einer spezifizierten Datei in der Zeile der #include-Direktive in ein Programm ein. Als Konvention wird die einzuf.gende Datei mit Header-Datei bezeichnet. Header-Dateien k.nnen beliebige Anweisungen und Pre-Prozessor- Direktiven enthalten. Als Konvention besitzen alle Header-Dateien von Clipper die Datei-namenserweiterung .ch.
Um zu entscheiden, wo Header-Dateien abzulegen sind, bestehen grunds.tz-lich zwei M.glichkeiten. Zum einen k.nnen sie im gleichen Verzeichnis wie die Quelldatei abgelegt werden, wenn sie lokal f.r ein bestimmtes Programm zur Verf.gung stehen sollen. Sollen sie hingegen global verf.gbar sein, so k.nnen sie in dem Verzeichnis abgelegt werden, das .ber die DOS-Umgebungsvariable INCLUDE spezifiziert wurde. In dieser Variable d.rfen gleichzeitig mehrere Verzeichnisse angegeben werden.

Header-Dateien r.umen einen wesentlichen Nachteil bei der Definition von Pseudo-Funktionen und Konstanten aus. Alle in einer solchen Datei vorgenommenen Definitionen gelten nur f.r die Datei, innerhalb derer die #include-Datei angegeben wurde. Der Pre-Prozessor f.gt solche Dateien dann in die jeweilige Programmdatei ein, bevor sie compiliert wird. Damit stehen alle einmalig vorgenommenen Definitionen im jeweils gew.nschten Programm zur Verf.gung.

Die Header-Datei Inkey.ch enth.lt besipielsweise eine Reihe von Definitionen f.r Tastencodes. Anstatt alle Direktiven dieser Datei als Dateikopf in jeder Programmdatei anzugeben, k.nnen diese durch Angabe der #include-Direktive am Anfang des jeweiligen Quellprogrammes eingebunden werden:

```
#include "Inkey.ch"
```

Ist diese Vereinbarung getroffen worden, sucht der Pre-Prozessor nach der Datei Inkey.ch und f.gt alle dort enthaltenen Direktiven am Anfang des jeweiligen Quellprogrammes ein.

Ein anderer Vorteil in der Verwendung der #include-Direktive ist, da. alle Definitionen in einer Datei enthalten sind. Sind .nderungen an diesen Definitionen notwendig, brauchen diese nur in der mit der #include-Direktive angegebenen Datei zu erfolgen, die eigentlichen Quellprogramme sind davon nicht betroffen.

Beachten Sie bitte, da. der G.ltigkeitsbereich von Definitionen innerhalb der eingef.gten Header-Datei auf die Programmdatei eingeschr.nkt ist, in der eine #include-Direktive angegeben wurde. Dies

gilt allerdings nicht, wenn der Compiler-Schalter /U innerhalb der DOS-Kommandozeile angegeben wird. In diesem Falle gelten Definitionen aus #include-Dateien f.r alle Programmdateien, die w.hrend einer Compilation aufgerufen werden.

Hinweise

. Unterst.tzende Header-Dateien: Clipper 5.0 wird durch eine Anzahl von Header-Da-teien unterst.tzt, die Konstanten-Definitionen f.r gemeinsame Operationen enthalten. Diese Dateien werden bei der Installation im Verzeichnis CLIPPER5/INCLUDE abgelegt.

. STD.CH - Standard Header-Datei: STD.CH ist die von Clipper 5.0 bereitgestellte Standard Header-Datei. Sie wird normalerweise vom Installationsprogramm im Verzeichnis CLIPPER5\INCLUDE abgelegt. Die Header-Datei STD.CH enth.lt die Definitionen f.r alle Clipper 5.0-Befehle sowie f.r einige Standard-Funktionen, die als Pseudo-Funktionen definiert wurden.
Die Standard Header-Datei weicht von den Header-Dateien ab, die normalerweise mit #include einbezogen werden. Erstens kann sie keinen anderen Code als Direktiven und EXTERNAL-Anweisungen enthalten. Zweitens ist alles, was in dieser Datei definiert wurde, f.r alle Programm-Dateien einer Compilation, und nicht nur f.r eine bestimmte, g.ltig.

. Um Definitionen global verf.gbar zu machen, sollten Sie in jedem Fall die #include-Direktive benutzen und so Header-Dateien mit gemeinsamen Definitionen einbeziehen. Als generelle Regel kann gelten, da. am Anfang aller Programmdateien explizit eine #include-Direktive anzugeben ist. Dies erm.glicht eine wesentlich einfachere Verfolgung der verwendeten Bezeichner.

Beispiele

Das folgende Beispiel demonstriert die Verwendung einer #include-Direktive, um die Datei mit Tastencode-Definitionen (Inkey.ch) in ein Programm einzubinden:

```
#include "Inkey.ch"

FUNCTION GetEvent()
  LOCAL nKey := 0, nResult

  DO CASE
  CASE nKey = K_F10
    nResult := DoMenu("Browse")
  CASE nKey = K_ESC
    nResult := DoQuit("Verlassen J/N ? ")
  CASE nKey = K_CTRL_RIGHT
    nResult := DoNextRec()
  ENDCASE
RETURN nResult
```

See Also:

#undef Hebt eine #define Makro-Definition wieder auf

Syntax

#undef <Bezeichner>

Parameter

<Bezeichner> ist der als ung.ltig zu erkl.rende Name einer symbolischen Konstante oder Pseudo-Funktion.

Anwendung

#undef hebt einen mit der #define-Direktive definierten Bezeichner wieder auf. Nach einer #undef-Direktive wird der dort angegebene Bezeichner ung.ltig. Die #undef-Direktive kann dazu benutzt werden, eine Definition zu beseitigen, bevor eine Neudefinition erfolgt. Dies vermeidet Compiler-Warnungen, die ausgegeben werden, falls die Redefinition eines Bezeichners erfolgt. Die #undef-Direktive kann auch verwendet werden, damit eine bedingte Compilation nur in bestimmten Teilen eines Programmes G.ltigkeit hat.

Beispiele

. Dieses Beispiel zeigt die Definition einer symbolischen Konstante und einer Pseudo-Funktion mit anschlie.ender #undef-Direktive:

```
  #define K_ESC 27
  #define MAX(x, y) (IF(x > y, x, y))
  .
  .<Anweisungen>...
  .
  #undef K_ESC
  #undef MAX
```

. Dieses Beispiel verwendet #undef, um einen Bezeichner vor einer Neudefinition zu entfernen, und so eine Compiler-Warnung zu vermeiden:

```
  #define DEBUG
  .
  .<Anweisungen>...
  .
  #undef DEBUG
  #define DEBUG .T.
```

. In diesem Beispiel wird ein Bezeichner entfernt, falls dieser bereits existiert. Anderenfalls wird er f.r weitere Teile des Quellprogrammes definiert:

```
  #ifdef TEST
    #undef TEST
  #else
    #define TEST
  #endif
```

See Also:

[^^Up^^]

Pr.zedenz der Operatoren

Werden Ausdr.cke evaluiert, die zwei oder mehr Operationen enthalten, und die nicht explizit mit Klammern gruppiert sind, so h.lt sich Clipper an einen definierten Vorrat von Regeln, um die Pr.zedens (Rang- oder Reihenfolge) der einzelnen Operationen festzulegen. Diese Regeln, sogenannte Pr.zedens-Regeln, definieren die Hierarchie aller Operatoren.

Achtung
Alle Funktionsaufrufe, bzw. alle speziellen Operatoren werden vor jeder anderen Operation evaluiert.

Pr.zedens der verschiedenen Kategorien

Verwendet man mehr als einen Operatortyp in einem Ausdruck, werden zuerst alle Teilausdr.cke in einem Pr.zedens-Level evaluiert, bevor die Teilausdr.cke der n.chsten Stufe ausgef.hrt werden. Mit Ausnahme von mehreren In-Line Zuweisungen, die von rechts nach links abgearbeitet werden, findet die Ausf.hrung von Teilausdr.cken immer von links nach rechts statt. Die Reihenfolge der Operatoren ist dabei wie folgt festgelegt:

1. Pre-Inkrement und Pre-Dekrement
2. Zeichenketten-Operationen
3. Datumswert-Operationen
4. Mathematische Operationen
5. Relationale Operationen
6. Logische Operationen
7. Zuweisungen
8. Post-Inkrement und Post-Dekrement

Der nicht zuweisende Teil einer kombinierten Zuweisung (z.B. *=) geh.rt je nach Operation in die Stufen 2, 3 oder 4, der zuweisende Teil des Operators hingegen in die Stufe 7.

Pr.zedens innerhalb einer Kategorie

Um zu garantieren, da. ohne explizite Klammerung identische Ausdr.cke die gleichen Resultate erzielen, existiert unter Clipper eine festgelegte Reihenfolge der Operatoren:

Pre-Inkrement und Pre-Dekrement
Die Prefix-Versionen der Inkrement- und Dekrement-Operatoren (++ / --) werden von links nach rechts evaluiert und besitzen die gleiche Pr.zedens-Stufe.

Zeichenketten
Die Verkn.pfungs-Operatoren (+ und -) f.r Zeichenketten werden von links nach rechts evaluiert und besitzen die gleiche Pr.zedens-Stufe.

Datum
Der einzige zul.ssige Operator f.r Datumswerte (gleiche Kategorie), die Subtraktion (-), wird von links nach rechts evaluiert.

Mathematische Operatoren
Besteht ein mathematischer Ausdruck aus mehr als einem Operator, werden zuerst alle Teilausdr.cke der gleichen Pr.zedens-Stufe evaluiert, bevor die Teilausdr.cke der n.chsten Stufe berechnet werden. Operationen einer Stufe werden von links nach rechts ausgef.hrt. Die Rangfolge bei mathematischen Operatoren ist wie folgt festgelegt:

1. Un.r Positiv oder Negativ (Vorzeichen + oder i)
2. Exponentation (oder **)
3. Multiplikation, Division und Modulus (* oder / oder %)
4. Addition und Subtraktion (+ und -)

Relationale Operatoren

Alle relationalen Operatoren besitzen die gleiche Rangfolge und werden von links nach rechts berechnet.

Logische Operatoren

Wie bei mathematischen Operatoren besitzen auch die logischen Operatoren eine vorgegebene Rangfolge. Besteht ein Ausdruck aus mehr als einem Operator, werden zuerst alle Teilausdr.cke der gleichen Pr.zedens-Stufe berechnet. Operationen einer Stufe werden von links nach rechts ausgef.hrt. Die Rangfolge bei logischen Operatoren ist wie folgt festgelegt:

1. Un.re Negation (.not. oder !)
2. Logisches UND (.and.)
3. Logisches ODER (.or.)

Zuweisung

Alle Zuweisungsoperatoren besitzen die gleiche Rangfolge und werden von rechts nach links berechnet. Bei kombinierten Operatoren (+=) wird der nicht zuweisende Teil (z.B. Addition) der Operation zuerst ausgef.hrt. Die Zuweisungen durch den Inkrement- und Dekrement-Operator besitzen eine eigene Pr.zedens-Stufe und fallen nicht unter diese Kategorie.

Post-Inkrement und Post-Dekrement

Die Postfix-Versionen der Inkrement- und Dekrement-Operatoren (++ und --) werden von links nach rechts berechnet und besitzen die gleiche Rangfolge.

Klammerung

Die normale Reihenfolge der Ausf.hrung kann durch explizite Klammerung ver.ndert werden. Sind in einem Ausdruck Klammern enthalten, wird der in diesen Klammern eingeschlossene Teilausdruck zuerst berechnet, wobei die Pr.zedens-Regeln eingehalten werden. Sind Klammerungen geschachtelt, beginnt die Abarbeitung mit der innersten Ebene der Klammerung.

```
%          Modulus zweier num. Werte -- bin.r              (mathematisch)
-------------------------------------------------------------------------
```

Syntax

<nExp1> % <nExp1>

Typ

numerisch

Operanden

<nExp1> ist der Dividend der Division.

<nExp2> ist der Divisor der Division.

Beschreibung

Die Modulus Operation liefert den Rest einer Division von <nExp1> / <nExp2>.

Hinweis

Ist der Divisor 0, wird ein Laufzeitfehler (Runtime-Error) erzeugt.

Beispiele

```
? 3 % 0                    // Resultat: Laufzeitfehler
? 3 % -2                   // Resultat: 1
? -3 % 2                   // Resultat: -1
? -3 % -2                  // Resultat: 1
? -3 % 0                   // Resultat: Laufzeitfehler
```

```
*          Multiplikation            -- bin.r              (mathematisch)
-------------------------------------------------------------------------
```

Syntax

<nExp1> * <nExp2>

Typ

numerisch

Operanden

<nExp1> und <nExp2> sind die zu multiplizierenden, numerischen Ausdr.cke.

Beschreibung

Der Multiplikations-Operator (*) ist ein bin.rer Operator, welcher das numerische Ergebnis einer Multiplikation der Werte <nExp1> und <nExp2> liefert.

Beispiele

```
? 3 * 0                    // Resultat: 0
? 3 * 2                    // Resultat: 6
? 3 * -2                   // Resultat: -6
? -3 * -2                  // Resultat: 6
? -3 * 0                   // Resultat: 0
? -1 * 3                   // Resultat: -3
? -2 * 3                   // Resultat: -6
? 0 * 3                    // Resultat: 0
```

**, ^ Exponentation -- bin.r (mathematisch)

Syntax

<nExp1> ** <nExp2>
<nExp1> ^ <nExp2>

Typ

numerisch

Operanden

<nExp1> ist die Mantisse.

<nExp2> ist der Exponent.

Beschreibung

Der Exponential-Operator (** oder ^) ist ein bin.rer Operator, welcher das numerische Ergebnis einer Exponentation mit der Mantisse <nExp1> und dem Exponent <nExp2> liefert.

Beispiele

```
SET DECIMALS TO 16
?  3 **  0                    // Resultat: 1.000000000
?  3 **  1                    // Resultat: 3.000000000
?  3 ** -2                    // Resultat: 0.111111111
? -3 **  2                    // Resultat: 9.000000000
? -3 **  0                    // Resultat: 1.000000000
? -1 **  3                    // Resultat: 1.000000000
? -2 **  3                    // Resultat: 8.000000000
?  2 ** -3                    // Resultat: 0.125000000
?  1 ** -3                    // Resultat: 1.000000000
```

+ Addition, un.r positiv, Zusammenf.gung (math, Zeichen)

Syntax

<nExp1> + <nExp2>
<dExp> + <nExp3>
<cExp1> + <cExp2>

Typ

Numerisch, Datum, Zeichen und Memo

Operanden

<nExp1> ist ein numerischer Wert, der um <nExp2> erh.ht werden soll.

<dExp> ist ein Datum, das um <nExp3> Tage erh.ht werden soll.

<cExp1> ist eine Zeichenkette, an die die Zeichenkette <cExp2> angeh.ngt werden soll.

Beschreibung

Der Additions-Operator (+) erm.glicht eine Reihe von verschiedenen Operationen, die vom Typ der Operanden abh.ngig sind:

. Positives Vorzeichen numerischer Werte: Das Plus-Zeichen (+) als Vorzeichen vor einem numerischen Wert f.hrt keine Operation aus, sondern kennzeichnet diesen als positiven Wert und ordnet eine h.here Rangfolge (Pr.zedens) zu.

. Addition zweier numerischer Werte: Sind beide Operanden von Typ numerisch, f.hrt der Plus-Operator (+) eine Addition der Operanden durch und liefert das Ergebnis dieser Operation als numerischen Wert.

. Addition eines Datums mit einem numerischen Wert: Ist einer der Operanden vom Typ Datum, mu. der zweite Operand vom Typ numerisch sein. Die Operation addiert zu diesem Datum die Anzahl Tage, die durch den numerischen Operanden beschrieben werden. Das Ergebnis ist vom Typ Datum, wobei Monatswechsel, Jahreswechsel und Schaltjahre ber.cksichtigt werden.

. Zusammenf.gung zweier Zeichenketten: Sind beide Operanden vom Typ Zeichen, wird der rechte Operand (cExp2) an den linken Operand (cExp1) angeh.ngt und als Ergebnis dieser Operation geliefert.

Beispiele

. Addition zweier numerischer Werte:

```
? 1 + 1                    // Resultat: 2
? 3 + 0                    // Resultat: 3
? 3 + 2                    // Resultat: 5
? -1 + 2                   // Resultat: 1
```

. Addition eines Datums mit einem numerischen Wert:

```
SET DATE GERMAN
? CTOD("01.01.90") + 2     // Resultat: 03.01.90
? CTOD("27.01.90") + 10    // Resultat: 06.02.90
? 10 + CTOD("31.12.90")    // Resultat: 10.01.91
```

. Zusammenf.gung zweier Zeichenketten:

```
? "Hallo " + "Ihr"         // Resultat: "Hallo Ihr"
? "**" + "Nantucket" + "**"    // Resultat: "**Nantucket**"
```

- Subtraktion, un.r negativ, Zusammenf.gung (math, Zeichen)

Syntax

<nExp1> - <nExp2>
<dExp1> - <nExp3>
<dExp2) - <dExp3>
<cExp1> - <cExp2>

Typ

Numerisch, Datum, Zeichen und Memo

Operanden

<nExp1> ist ein numerischer Wert, der um <nExp2> erniedrigt werden soll.

<dExp1> ist ein Datum, das um <nExp3> Tage erniedrigt werden soll.

<dExp2> subtrahiert mit <dExp3> liefert die Datumsdifferenz in Tagen.

<cExp1> ist eine Zeichenkette, an die die Zeichenkette <cExp2> angeh.ngt werden soll, wobei Leerzeichen am Ende von <cExp1> nicht ber.cksichtigt werden.

Beschreibung

Der Subtraktions-Operator (-) erm.glicht eine Reihe von verschiedenen Operationen, die vom Typ der Operanden abh.ngig sind:

. Negatives Vorzeichen numerischer Werte: Das Minus-Zeichen (-) als Vorzeichen vor einem numerischen Wert kennzeichnet diesen als negativen Wert durch die Multiplikation mit -1.

. Subtraktion zweier numerischer Werte: Sind beide Operanden von Typ numerisch, f.hrt der Minus-Operator (-) eine Subtraktion der Operanden durch und liefert das Ergebnis dieser Operation als numerischen Wert.

. Subtraktion eines Datums mit einem numerischen Wert: Ist einer der Operanden vom Typ Datum, mu. der zweite Operand vom Typ numerisch sein. Die Operation subtrahiert von diesem Datum die Anzahl Tage, die durch den numerischen Operanden beschrieben werden. Das Ergebnis ist vom Typ Datum, wobei Monatswechsel, Jahreswechsel und Schaltjahre ber.cksichtigt werden.

. Subtraktion zweier Datums-Werte: Sind beide Operanden vom Typ Datum, so werden beide Datumswerte voneinander abgezogen und die Differenz in Tagen als numerisches Ergebnis geliefert.

. Zusammenf.gung zweier Zeichenketten: Sind beide Operanden vom Typ Zeichen, wird der rechte Operand (cExp2) an den linken Operanden (cExp1) angeh.ngt und als Ergebnis dieser Operation geliefert. Sind im linken Operanden (cExp1) Leerzeichen am Ende enthalten, so werden diese nicht ber.cksichtigt.

Beispiele

. Subtraktion zweier numerischer Werte:

```
? 1 - 1                    // Resultat: 0
? 3 - 0                    // Resultat: 3
? 3 - 2                    // Resultat: 1
? -1 - 2                   // Resultat: -3
```

. Subtraktion eines Datums mit einem numerischen Wert:

```
SET DATE GERMAN
? CTOD("01.01.90") - 2     // Resultat: 30.12.89
```

```
  ? CTOD("27.01.90") - 10         // Resultat: 17.01.90
  ? 10 - CTOD("31.12.90")         // Resultat: Laufzeitfehler
```

. Subtraktion zweier Datumswerte:

```
  SET DATE GERMAN
  ? CTOD("31.01.90") - CTOD("30.01.90") // Resultat: 1
  ? CTOD("31.01.90") - CTOD("01.01.90") // Resultat: 30
  ? CTOD("01.01.90") - CTOD("31.01.90") // Resultat: -30
```

. Zusammenf.gung zweier Zeichenketten:

```
  ? "Hallo " - "Ihr"              // Resultat: "HalloIhr"
  ? "**" + "Nantucket" + "**"     // Resultat: "**Nantucket**"
```

/ Division -- bin.r (mathematisch)

Syntax

<nExp1> / <nExp2>

Typ

Numerisch

Operanden

<nExp1> ist der Divident der Operation.
<nExp2> ist der Divisor der Operation.

Beschreibung

Der Divisions-Operator (/) f.hrt eine Divisionsoperation mit dem Divident (<nExp1>) und dem Divisor (<nExp2>) durch und liefert das Ergebnis als numerischen Wert.

Hinweis

Besitzt der Divisor (<nExp2>) den Wert 0, f.hrt die Operation zu einem Laufzeitfehler.

Beispiele

```
SET DECIMALS TO 16
?  3 /  0                    // Resultat: Laufzeitfehler
?  3 /  2                    // Resultat:  1.500000000
?  3 / -2                    // Resultat: -1.500000000
? -2 / -3                    // Resultat:  0.666666667
? -3 /  0                    // Resultat: Laufzeitfehler
? -1 /  3                    // Resultat: -0.333333333
?  9 /  3                    // Resultat:  3.000000000
?  0 /  3                    // Resultat: ********** .berlauf
```

<>,#,!= Ungleich -- bin.r (relational)

Syntax

<Exp1> <> <Exp2>
<Exp1> # <Exp2>
<Exp1> != <Exp2>

Typ

Logisch, Numerisch, Datum, Zeichen, Datum, NIL

Operanden

<Exp1> und <Exp2> sind die zu vergleichenden Operanden, die gleichen Typs sein m.ssen, mit Ausnahme von NIL.

Beschreibung

Die Vergleichsoperatoren <>, # und != pr.fen die Operanden auf Ungleichheit, die beide den gleichen Typ besitzen m.ssen. Das Ergebnis der Operation liefert einen logischen Wert, der den Inhalt TRUE (.T.) besitzt, wenn die Operatoren nicht identisch sind, ansonsten wird FALSE (.F.) zur.ckgegeben.

Hinweis

Sind beide Operatoren vom Typ Zeichen, basiert die Vergleichsoperation auf den ASCII-Werten der Zeichen innerhalb der Zeichenketten. Das linke Zeichen innerhalb der Zeichenkette besitzt dabei eine h.here Priorit.t wie das rechte Zeichen. Die Operation ber.cksichtigt dabei die Einstellung des Clipper-Schalters SET EXACT ON/OFF.

Beispiele

. Vergleich logischer Werte:

```
? .T. <> .T.                  // Resultat: .F.
? .T. <> .F.                  // Resultat: .T.
```

. Vergleich numerischer Werte:

```
?  1  <>  1                   // Resultat: .F.
?  2  <>  1                   // Resultat: .T.
```

. Vergleich von Datumswerten:

```
SET DATE GERMAN
? CTOD("01.01.90") <> CTOD("01.01.90")  // Resultat: .F.
? CTOD("31.01.90") <> CTOD("01.01.90")  // Resultat: .T.
```

. Vergleich von Zeichenketten:

```
SET EXACT OFF
? "Clipper" <> "Clipper"       // Resultat: .F.
? "Clipper" <> "Clipper 5.0"   // Resultat: .T.
? "Clipper 5.0" <> "Clipper"   // Resultat: .F.
? "Clipper" <> "Flipper"       // Resultat: .T.

SET EXACT ON
? "Clipper" <> "Clipper"       // Resultat: .F.
? "Clipper" <> "Clipper 5.0"   // Resultat: .T.
? "Clipper 5.0" <> "Clipper"   // Resultat: .T.
? "Clipper" <> "Flipper"       // Resultat: .T.
```

. Vergleich mit NIL:

```
? 1 <> NIL                     // Resultat: .T.
? NIL <> NIL                   // Resultat: .F.
```

$ Zeichenfolge enthalten in -- bin.r (relational)

Syntax

<cExp1> $ <cExp2>

Typ

Zeichen, Memo

Operanden

<cExp1> ist eine Teilzeichenkette, die in <cExp2> auf Vorhandensein gesucht wird.

Beschreibung

Der relationale Operator ($) pr.ft auf Vorkommen der Zeichen im linken Operanden (<cExp1>) im rechten Operanden (<cExp2>). Das Ergebnis der Operation ist TRUE (.T.), wenn <cExp1> in <cExp2> enthalten ist, ansonsten wird FALSE (.F.) geliefert.

Hinweis

Die Operation unterscheidet zwischen Gro.- und Kleinbuchstaben.

Beispiele

```
? "ip" $ "Clipper"              // Resultat: .T.
? "cl" $ "Clipper"              // Resultat: .F.
```

.AND. Logisches UND -- bin.r (logisch)

Syntax

<lExp1> .AND. <lExp2>

Typ

Logisch

Operanden

<lExp1> und <lExp2> sind logische Operanden.

Beschreibung

Der .AND. Operator ist ein bin.rer, logischer Operator, der eine logische UND-Verkn.pfung beider logischen Operanden vornimmt. Das Ergebnis dieser Operation liefert TRUE (.T.) wenn beide Operanden (<lExp1> und <lExp2>) den Wert TRUE enthalten, ansonsten ergibt die Operation den Wert FALSE (.F.).

Beispiele

```
?  .T. .AND. .T.                  // Resultat: .T.
?  .F. .AND. .T.                  // Resultat: .F.
?  .T. .AND. .F.                  // Resultat: .F.
?  .F. .AND. .F.                  // Resultat: .F.
```

.NOT.,! Logische Negation -- bin.r (logisch)

Syntax

.NOT. <lExp>
! <lExp>

Typ

Logisch

Operanden

<lExp> ist der zu negierende, logische Operand.

Beschreibung

Der .NOT. Operator ist ein un.rer, logischer Operator, der als Ergebnis den umgekehrten Wert des Operanden liefert.

Beispiele

```
? ! .T.                          // Resultat: .F.
? ! .F.                          // Resultat: .T.
```

.OR. Logisches ODER -- bin.r (logisch)

Syntax

<lExp1> .OR. <lExp2>

Typ

Logisch

Operanden

<lExp1> und <lExp2> sind logische Operanden.

Beschreibung

Der .OR. Operator ist ein bin.rer, logischer Operator, der eine logische ODER-Verkn.pfung beider logischen Operanden vornimmt. Das Ergebnis dieser Operation liefert TRUE (.T.), wenn einer der Operanden (<lExp1> oder <lExp2>) den Wert TRUE enth.lt, ansonsten ergibt die Operation den Wert FALSE (.F.).

Beispiele

```
?  .T. .OR. .T.                  // Resultat: .T.
?  .F. .OR. .T.                  // Resultat: .T.
?  .T. .OR. .F.                  // Resultat: .T.
?  .F. .OR. .F.                  // Resultat: .F.
```

```
<          Kleiner                    -- bin.r                 (relational)
--------------------------------------------------------------------------
```

Syntax

<Exp1> < <Exp2>

Typ

Logisch, Numerisch, Datum, Zeichen und Memo

Operanden

<Exp1> und <Exp2> sind die zu vergleichenden Operanden, die gleichen Typs sein m.ssen.

Beschreibung

Der Kleiner-Operator (<) ist ein relationaler Vergleichsoperator. Er vergleicht zwei Operanden. Das logische Ergebnis der Operation ist TRUE, wenn <Exp1> kleiner als <Exp2> ist. Die Wertigkeit der Operanden ist vom Typ abh.ngig.

. Logisch: FALSE (.F.) besitzt eine geringere Wertigkeit als TRUE (.T.).

. Numerisch: Die Wertigkeit wird bestimmt durch den Wert der Operanden.

. Datum: Das j.ngere Datum hat eine geringere Wertigkeit als das .ltere Datum.

. Zeichen: Die Wertigkeit wird bestimmt durch die ASCII-Codes der Zeichen. Bei Zeichenketten beginnt die Vergleichsoperation mit dem linken Zeichen innerhalb der Operanden.

"A" (ASCII-Code 65) ist kleiner als "B" (ASCII-Code 66).

Beispiele

. Logisch:

```
?  .T. < .T.                  // Resultat: .F.
?  .F. < .T.                  // Resultat: .T.
?  .T. < .F.                  // Resultat: .F.
?  .F. < .F.                  // Resultat: .F.
```

. Numerisch:

```
?   1  <  3                   // Resultat: .T.
?   3  <  1                   // Resultat: .F.
?   3  <  3                   // Resultat: .F.
?  -3  <  1                   // Resultat: .T.
?  -1  < -3                   // Resultat: .F.
```

. Datum:

```
SET DATE GERMAN
? CTOD("01.01.90") < CTOD("31.01.90")  // Resultat: .T.
? CTOD("01.01.90") < CTOD("01.01.90")  // Resultat: .F.
? CTOD("31.01.90") < CTOD("01.01.90")  // Resultat: .F.
```

. Zeichen:

```
? "A" < "B"                   // Resultat: .T.
? "B" < "A"                   // Resultat: .F.
? "A" < "A"                   // Resultat: .F.
? "a" < "A"                   // Resultat: .F.
? "A" < "a"                   // Resultat: .T.
```

ASIZE() Vergr..ert oder verkleinert ein Array

Syntax

ASIZE(<aArray>, <nL.nge>) -> aArray

Parameter

<aArray> ist das Array, das vergr..ert oder verkleinert werden soll.

<nL.nge> ist die neue Gr..e des Arrays.

R.ckgabe

ASIZE() liefert eine Referenz auf <aArray>.

Anwendung

ASIZE ist eine Arrayfunktion, welche die aktuelle Gr..e des Arrays <aArray> .ndert. Das Array kann durch die Angabe in <nL.nge> verk.rzt oder verl.ngert werden. Wird ein Array verk.rzt, gehen die Elemente am Ende des Arrays verloren. Wird das Array verl.ngert, werden neue Elemente vom Typ NIL an das Ende des Arrays angef.gt.

ASIZE() ist vergleichbar mit AADD(), dort kann aber lediglich ein Element angef.gt werden. AADD() kann jedoch zur gleichen Zeit dem neuen Element einen Wert zuweisen. Beachten Sie, da. ein Unterschied zu den Funktionen AINS() und ADEL() besteht, welche die aktuelle L.nge der Arrays nicht .ndern.

Beispiele

Diese Beispiele demonstrieren das Anf.gen neuer Elemente und das L.schen von Elementen:

```
aArray := { 1 }      // Resultat: { 1 }
ASIZE(aArray, 3)     // Resultat: { 1, NIL, NIL }
ASIZE(aArray, 2)     // Resultat: { 1, NIL }
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

```
<=          Kleiner oder gleich       -- bin.r                 (relational)
-------------------------------------------------------------------------------
```

Syntax

<Exp1> <= <Exp2>

Typ

Logisch, Numerisch, Datum, Zeichen und Memo

Operanden

<Exp1> und <Exp2> sind die zu vergleichenden Operanden, die gleichen Typs sein m.ssen.

Beschreibung

Der Kleiner-Gleich Operator (<=) ist ein relationaler Vergleichsoperator. Er vergleicht zwei Operanden. Das logische Ergebnis der Operation ist TRUE, wenn <Exp1> kleiner oder gleich <Exp2> ist. Die Wertigkeit der Operanden ist vom Typ abh.ngig.

. Logisch: FALSE (.F.) besitzt eine geringere Wertigkeit als TRUE (.T.).

. Numerisch: Die Wertigkeit wird bestimmt durch den Wert der Operanden.

. Datum: Das j.ngere Datum hat eine geringere Wertigkeit als das .ltere Datum.

. Zeichen: Die Wertigkeit wird bestimmt durch die ASCII-Code der Zeichen. Bei Zeichenketten beginnt die Vergleichsoperation mit dem linken Zeichen innerhalb der Operanden.

"A" (ASCII-Code 65) ist kleiner als "B" (ASCII-Code 66).

Beispiele

. Logisch:

```
? .T. <= .T.                  // Resultat: .T.
? .F. <= .T.                  // Resultat: .T.
? .T. <= .F.                  // Resultat: .F.
? .F. <= .F.                  // Resultat: .T.
```

. Numerisch:

```
?   1  <=  3                  // Resultat: .T.
?   3  <=  1                  // Resultat: .F.
?   3  <=  3                  // Resultat: .T.
?  -3  <=  1                  // Resultat: .T.
?  -1  <= -3                  // Resultat: .F.
```

. Datum:

```
SET DATE GERMAN
? CTOD("01.01.90") <= CTOD("31.01.90")  // Resultat: .T.
? CTOD("01.01.90") <= CTOD("01.01.90")  // Resultat: .T.
? CTOD("31.01.90") <= CTOD("01.01.90")  // Resultat: .F.
```

. Zeichen:

```
? "A" <= "B"                  // Resultat: .T.
? "B" <= "A"                  // Resultat: .F.
? "A" <= "A"                  // Resultat: .T.
? "a" <= "A"                  // Resultat: .F.
? "A" <= "a"                  // Resultat: .T.
```

```
=          Gleich                    -- bin.r                 (relational)
-------------------------------------------------------------------------
```

Syntax

<Exp1> = <Exp2>

Typ

Logisch, Numerisch, Datum, Zeichen und Memo

Operanden

<Exp1> und <Exp2> sind die zu vergleichenden Operanden, die gleichen Typs sein m.ssen.

Beschreibung

Der Gleich-Operator (=) ist ein relationaler Vergleichsoperator. Er vergleicht zwei Operanden. Das logische Ergebnis der Operation ist TRUE, wenn <Exp1> gleich <Exp2> ist. Die Wertigkeit der Operanden ist vom Typ abh.ngig.

. Logisch: FALSE (.F.) besitzt eine geringere Wertigkeit als TRUE (.T.).

. Numerisch: Die Wertigkeit wird bestimmt durch den Wert der Operanden.

. Datum: Das j.ngere Datum hat eine geringere Wertigkeit als das .ltere Datum.

. Zeichen: Die Wertigkeit wird bestimmt durch die ASCII-Codes der Zeichen. Bei Zeichenketten beginnt die Vergleichsoperation mit dem linken Zeichen innerhalb der Operanden.

"A" (ASCII-Code 65) ist kleiner als "B" (ASCII-Code 66).

Bei unterschiedlich langen Zeichenketten ist das Ergebnis abh.ngig von der Clipper-Schaltereinstellung SET EXACT.

Beispiele

. Logisch:

```
?  .T. = .T.                  // Resultat: .T.
?  .F. = .T.                  // Resultat: .F.
?  .T. = .F.                  // Resultat: .F.
?  .F. = .F.                  // Resultat: .T.
```

. Numerisch:

```
?   1  =  3                   // Resultat: .F.
?   3  =  1                   // Resultat: .F.
?   3  =  3                   // Resultat: .T.
?  -3  =  1                   // Resultat: .F.
?  -1  = -3                   // Resultat: .F.
```

. Datum:

```
SET DATE GERMAN
? CTOD("01.01.90") = CTOD("31.01.90")  // Resultat: .F.
? CTOD("01.01.90") = CTOD("01.01.90")  // Resultat: .T.
? CTOD("31.01.90") = CTOD("01.01.90")  // Resultat: .F.
```

. Zeichen:

```
? "A" = "B"                   // Resultat: .F.
? "B" = "A"                   // Resultat: .F.
? "A" = "A"                   // Resultat: .T.
? "a" = "A"                   // Resultat: .F.
```

```
? "A" = "a"                     // Resultat: .F.

SET EXACT OFF
? "Clipper" = "Clipper"         // Resultat: .T.
? "Clipper" = "Clipper 5.0"     // Resultat: .F.
? "Clipper 5.0" = "Clipper"     // Resultat: .T.
? "Clipper" = "Flipper"         // Resultat: .F.

SET EXACT ON
? "Clipper" = "Clipper"         // Resultat: .T.
? "Clipper" = "Clipper 5.0"     // Resultat: .F.
? "Clipper 5.0" = "Clipper"     // Resultat: .F.
? "Clipper" = "Flipper"         // Resultat: .F.
```

```
==          Exakt gleich              -- bin.r                 (relational)
--------------------------------------------------------------------------
```

Syntax

<Exp1> == <Exp2>

Typ

Logisch, Numerisch, Datum, Zeichen und Memo

Operanden

<Exp1> und <Exp2> sind die zu vergleichenden Operanden, die gleichen Typs sein m.ssen.

Beschreibung

Der Exakt-Gleich Operator (==) ist ein relationaler Vergleichsoperator. Er vergleicht zwei Operanden. Das logische Ergebnis der Operation ist TRUE, wenn <Exp1> gleich <Exp2> ist. Die Wertigkeit der Operanden ist vom Typ abh.ngig. Mit Ausnahme von Zeichenkettenvergleichen ist der Operator identisch mit dem Vergleichsoperator (=).

. Logisch: FALSE (.F.) besitzt eine geringere Wertigkeit als TRUE (.T.).

. Numerisch: Die Wertigkeit wird bestimmt durch den Wert der Operanden.

. Datum: Das j.ngere Datum hat eine geringere Wertigkeit als das .ltere Datum.

. Zeichen: Die Wertigkeit wird bestimmt durch die ASCII-Codes der Zeichen. Bei Zeichenketten beginnt die Vergleichsoperation mit dem linken Zeichen innerhalb der Operanden.

"A" (ASCII-Code 65) ist kleiner als "B" (ASCII-Code 66).

Der == Operator vergleicht bei Zeichenketten auf exakte .bereinstimmung von L.nge und Wertigkeit der Operanden. Die Einstellung des Clipper-Schalters SET EXACT wird bei dieser Operation nicht ber.cksichtigt.

Beispiele

. Logisch:

```
?  .T. == .T.                  // Resultat: .T.
?  .F. == .T.                  // Resultat: .F.
?  .T. == .F.                  // Resultat: .F.
?  .F. == .F.                  // Resultat: .T.
```

. Numerisch:

```
?   1  ==  3                   // Resultat: .F.
?   3  ==  1                   // Resultat: .F.
?   3  ==  3                   // Resultat: .T.
?  -3  ==  1                   // Resultat: .F.
?  -1  == -3                   // Resultat: .F.
```

. Datum:

```
SET DATE GERMAN
? CTOD("01.01.90") == CTOD("31.01.90")  // Resultat: .F.
? CTOD("01.01.90") == CTOD("01.01.90")  // Resultat: .T.
? CTOD("31.01.90") == CTOD("01.01.90")  // Resultat: .F.
```

. Zeichen:

```
? "A" == "B"                   // Resultat: .F.
? "B" == "A"                   // Resultat: .F.
```

```
? "A" == "A"                    // Resultat: .T.
? "a" == "A"                    // Resultat: .F.
? "A" == "a"                    // Resultat: .F.

SET EXACT OFF
? "Clipper" == "Clipper"        // Resultat: .T.
? "Clipper" == "Clipper 5.0"    // Resultat: .F.
? "Clipper 5.0" == "Clipper"    // Resultat: .F.
? "Clipper" == "Flipper"        // Resultat: .F.

SET EXACT ON
? "Clipper" == "Clipper"        // Resultat: .T.
? "Clipper" == "Clipper 5.0"    // Resultat: .F.
? "Clipper 5.0" == "Clipper"    // Resultat: .F.
? "Clipper" == "Flipper"        // Resultat: .F.
```

```
>          Gr..er                    -- bin.r                 (relational)
--------------------------------------------------------------------------
```

Syntax

<Exp1> > <Exp2>

Typ

Logisch, Numerisch, Datum, Zeichen und Memo

Operanden

<Exp1> und <Exp2> sind die zu vergleichenden Operanden, die gleichen Typs sein m.ssen.

Beschreibung

Der Gr..er-Operator (>) ist ein relationaler Vergleichsoperator. Er vergleicht zwei Operanden. Das logische Ergebnis der Operation ist TRUE, wenn <Exp1> gr..er als <Exp2> ist. Die Wertigkeit der Operanden ist vom Typ abh.ngig.

. Logisch: FALSE (.F.) besitzt eine geringere Wertigkeit als TRUE (.T.).

. Numerisch: Die Wertigkeit wird bestimmt durch den Wert der Operanden.

. Datum: Das j.ngere Datum hat eine geringere Wertigkeit als das .ltere Datum.

. Zeichen: Die Wertigkeit wird bestimmt durch die ASCII-Codes der Zeichen. Bei Zeichenketten beginnt die Vergleichsoperation mit dem linken Zeichen innerhalb der Operanden.

"A" (ASCII-Code 65) ist kleiner als "B" (ASCII-Code 66).

Beispiele

. Logisch:

```
?  .T. > .T.                  // Resultat: .F.
?  .F. > .T.                  // Resultat: .F.
?  .T. > .F.                  // Resultat: .T.
?  .F. > .F.                  // Resultat: .F.
```

. Numerisch:

```
?   1  >  3                   // Resultat: .F.
?   3  >  1                   // Resultat: .T.
?   3  >  3                   // Resultat: .F.
?  -3  >  1                   // Resultat: .F.
?  -1  > -3                   // Resultat: .T.
```

. Datum:

```
SET DATE GERMAN
? CTOD("01.01.90") > CTOD("31.01.90")  // Resultat: .F.
? CTOD("01.01.90") > CTOD("01.01.90")  // Resultat: .F.
? CTOD("31.01.90") > CTOD("01.01.90")  // Resultat: .T.
```

. Zeichen:

```
? "A" > "B"                   // Resultat: .F.
? "B" > "A"                   // Resultat: .T.
? "A" > "A"                   // Resultat: .F.
? "a" > "A"                   // Resultat: .T.
? "A" > "a"                   // Resultat: .F.
```

```
>=         Gr..er oder gleich        -- bin.r                 (relational)
------------------------------------------------------------------------------
```

Syntax

<Exp1> >= <Exp2>

Typ

Logisch, Numerisch, Datum, Zeichen und Memo

Operanden

<Exp1> und <Exp2> sind die zu vergleichenden Operanden, die gleichen Typs sein m.ssen.

Beschreibung

Der Gr..er-Gleich Operator (>=) ist ein relationaler Vergleichsoperator. Er vergleicht zwei Operanden. Das logische Ergebnis der Operation ist TRUE, wenn <Exp1> gr..er oder gleich <Exp2> ist. Die Wertigkeit der Operanden ist vom Typ abh.ngig.

. Logisch: FALSE (.F.) besitzt eine geringere Wertigkeit als TRUE (.T.).

. Numerisch: Die Wertigkeit wird bestimmt durch den Wert der Operanden.

. Datum: Das j.ngere Datum hat eine geringere Wertigkeit als das .ltere Datum.

. Zeichen: Die Wertigkeit wird bestimmt durch die ASCII-Codes der Zeichen. Bei Zeichenketten beginnt die Vergleichsoperation mit dem linken Zeichen innerhalb der Operanden.

"A" (ASCII-Code 65) ist kleiner als "B" (ASCII-Code 66).

Beispiele

. Logisch:

```
? .T. >= .T.                  // Resultat: .T.
? .F. >= .T.                  // Resultat: .F.
? .T. >= .F.                  // Resultat: .T.
? .F. >= .F.                  // Resultat: .T.
```

. Numerisch:

```
?  1  >=  3                   // Resultat: .F.
?  3  >=  1                   // Resultat: .T.
?  3  >=  3                   // Resultat: .T.
? -3  >=  1                   // Resultat: .F.
? -1  >= -3                   // Resultat: .T.
```

. Datum:

```
SET DATE GERMAN
? CTOD("01.01.90") >= CTOD("31.01.90")  // Resultat: .F.
? CTOD("01.01.90") >= CTOD("01.01.90")  // Resultat: .T.
? CTOD("31.01.90") >= CTOD("01.01.90")  // Resultat: .T.
```

. Zeichen:

```
? "A" >= "B"                  // Resultat: .F.
? "B" >= "A"                  // Resultat: .T.
? "A" >= "A"                  // Resultat: .T.
? "a" >= "A"                  // Resultat: .T.
? "A" >= "a"                  // Resultat: .F.
```

```
=          Einfache Zuweisung        -- bin.r                  (Zuweisung)
-------------------------------------------------------------------------------
```

Syntax

<idVar> = <Exp>

Typ

Alle

Operanden

<idVar> ist ein g.ltiger Bezeichner einer Clippervariablen (einschlie.lich Datenbankfeld).

<Exp> ist ein g.ltiger Ausdruck, dessen Wert der Variablen <idVar> zugewiesen werden soll.

Beschreibung

Der einfache Zuweisungsoperator wird ben.tigt, um einer Variablen oder einem Datenbankfeld ein Wert zuzuweisen. Er ist identisch mit dem STORE-Befehl, wenn er auf eine Variable angewendet wird.

Beispiele

. Anwendung auf Variablen:

```
nWert = 25
nWertNeu = nWert + 30
dDatumNeu = DATE() + nWert
```

. Die Anwendung des einfachen Zuweisungs-Operators auf Datenbankfelder

```
Kunden->Name = "M.ller"
// liefert das gleiche Resultat wie:
REPLACE Kunden->Name WITH "M.ller"
```

:= In-Line Zuweisung -- bin.r (Zuweisung)

Syntax

<idVar> := [<idVar2> := ...] <Exp>

Typ

Alle

Operanden

<idVar> ist ein g.ltiger Bezeichner einer Clippervariablen (einschlie.lich Datenbankfeld).

<Exp> ist ein g.ltiger Ausdruck, dessen Wert der Variablen <idVar> zugewiesen werden soll.

Beschreibung

Der In-Line Zuweisungsoperator (:=) kann genauso wie der einfache Zuweisungsoperator benutzt werden, um einer Variablen oder einem Datenbankfeld einen Wert zuzuweisen. Der In-Line Operator kann aber auch in jedem Befehl oder jeder Funktion auftauchen, deren Syntax den Einsatz eines Ausdrucks erlaubt.

Beispiele

. Einfache Wertzuweisung:

```
nZahl := 25
nNeueZahl := SQRT(nZahl) ** 5
nAlteZahl := nZahl
```

. Beispiele f.r In-Line Zuweisungen:

```
LOCAL nZahl := 10
IF (dDatum := DATE() - 30) < CTOD("01.01.90")
  ? SQRT( nZahl := (nZahl ** 2))
ENDIF

Kunden->ZahlBtrg := Rechnung->Betrag := RngBtrg := Netto * 1.14
```

+= Addition und Zuweisung -- bin.r (Zuweisung)

Syntax

<idVar> += <nExp>
<idcVar> += <cExp>

Typ

Numerisch, Zeichen und Memo

Operanden

<idVar> ist der Bezeichner einer existierenden und g.ltigen Clipper-Variablen vom Typ Datum oder Numerisch.

<nExp> ist numerischer Ausdruck, dessen Wert zu dem Wert in <idVar> addiert werden soll.

<idcVar> ist der Bezeichner einer existierenden und g.ltigen Clipper-Variablen vom Typ Zeichen.

<cExp> ist ein Zeichenkettenausdruck, der an die Zeichen-Variable <idcVar> angef.gt werden soll.

Beschreibung

Der kombinierte Zuweisungsoperator += addiert zuerst die Operanden und weist danach das Ergebnis der Operation dem linken Operanden zu.

Hinweis

Es ist zu beachten, da. dieser Operator ein In-Line-Operator ist und somit auch in jedem Befehl oder in jeder Funktion verwendet werden darf, deren Syntax den Einsatz eines Ausdrucks erlaubt.

Beispiele

```
nZahl := 10

? SQRT( nZahl += 15 )            // Resultat:  5
? nZahl                          // Resultat: 25
```

-= Subtraktion und Zuweisung -- bin.r (Zuweisung)

Syntax

<idVar> -= <nExp>
<idcVar> -= <cExp>

Typ

Numerisch, Datum, Zeichen und Memo

Operanden

<idVar> ist der Bezeichner einer existierenden und g.ltigen Clipper-Variablen vom Typ Datum oder Numerisch.

<nExp> ist numerischer Ausdruck, dessen Wert von dem Wert in <idVar> subtrahiert werden soll.

<idcVar> ist der Bezeichner einer existierenden und g.ltigen Clipper-Variable vom Typ Zeichen.

<cExp> ist ein Zeichenkettenausdruck, der an die Zeichen-Variable <idcVar> angef.gt werden soll.

Beschreibung

Der kombinierte Zuweisungsoperator -= g.ltig mit numerischen, Datums- und Zeichen-Datentypen. Bei numerischen Datentypen wird zuerst der Wert in <nExp> vom Wert in <idVar> subtrahiert und das Ergebnis in <idVar> abgelegt. Als Zeichen-Operator wird die Zeichenfolge in <cExp> an die Zeichenfolge in <idcVar> angef.gt, wobei Leerzeichen am Ende von <idcVar> unber.cksichtigt bleiben.

Hinweis

Es ist zu beachten, da. dieser Operator ein In-Line-Operator ist und somit auch in jedem Befehl oder in jeder Funktion verwendet werden darf, deren Syntax den Einsatz eines Ausdrucks erlaubt.

Beispiele

```
nZahl := 10

? SQRT( nZahl -= 5 ) ** 2         // Resultat: 25
? nZahl                           // Resultat:  5
```

*= Multiplikation und Zuweisung -- bin.r (Zuweisung)

Syntax

<idnVar> *= <nExp>

Typ

Numerisch

Operanden

<idnVar> ist der Bezeichner einer g.ltigen und numerisch initialisierten Clipper-Variable.

<nExp> ist ein numerischer Ausdruck, mit dem der Wert in <idnVar> multipliziert wird.

Beschreibung

Der kombinierte Zuweisungsoperator *= ist nur mit numerischen Operanden g.ltig. Es wird zun.chst eine Multiplikation durchgef.hrt, danach das Ergebnis in <idnVar> abgelegt.

Hinweis

Es ist zu beachten, da. dieser Operator ein In-Line-Operator ist und somit auch in jedem Befehl oder in jeder Funktion verwendet werden darf, deren Syntax den Einsatz eines Ausdrucks erlaubt.

Beispiele

```
nZahl := 10
? nNeueZahl := ( nZahl *= 5 ) + 100  // Resultat: 150
? nZahl                              // Resultat:  50
```

/= Division und Zuweisung -- bin.r (Zuweisung)

Syntax

<idnVar> /= <nExp>

Typ

Numerisch

Operanden

<idnVar> ist der Bezeichner einer g.ltigen und numerisch initialisierten Clipper-Variable.

<nExp> ist ein numerischer Ausdruck, durch den der Wert in <idnVar> dividiert wird.

Beschreibung

Der kombinierte Zuweisungsoperator /= ist nur mit numerischen Operanden g.ltig. Es wird zun.chst eine Division durchgef.hrt, danach das Ergebnis in <idnVar> abgelegt.

Hinweis

Es ist zu beachten, da. dieser Operator ein In-Line-Operator ist und somit auch in jedem Befehl oder in jeder Funktion verwendet werden darf, deren Syntax den Einsatz eines Ausdrucks erlaubt.

Beispiele

```
nZahl := 10
? nNeueZahl := ( nZahl /= 5 ) + 100  // Resultat: 105
? nZahl                              // Resultat:   5
```

%= Modulus und Zuweisung -- bin.r (Zuweisung)

Syntax

<idnVar> %= <nExp>

Typ

Numerisch

Operanden

<idnVar> ist der Bezeichner einer g.ltigen und numerisch initialisierten Clipper-Variable.

<nExp> ist ein numerischer Ausdruck, durch den der Wert in <idnVar> dividiert wird.

Beschreibung

Der kombinierte Zuweisungsoperator /= ist nur mit numerischen Operanden g.ltig. Es wird zun.chst eine Modulo-Operation durchgef.hrt, danach das Ergebnis in <idnVar> abgelegt.

Hinweis

Es ist zu beachten, da. dieser Operator ein In-Line-Operator ist und somit auch in jedem Befehl oder in jeder Funktion verwendet werden darf, deren Syntax den Einsatz eines Ausdrucks erlaubt.

Beispiele

```
nZahl := 10
? nNeueZahl := ( nZahl %= 4 ) * 100  // Resultat: 200
? nZahl                              // Resultat:   2
```

**=,^= Exponentation und Zuweisung -- bin.r (Zuweisung)

Syntax

<idnVar> ^= <nExp>
<idnVar> **= <nExp>

Typ

Numerisch

Operanden

<idnVar> ist der Bezeichner einer g.ltigen und numerisch initialisierten Clipper-Variable. Innerhalb der Operation bildet er die Mantisse.

<nExp> ist der Exponent.

Beschreibung

Der kombinierte Zuweisungsoperator ^= ist nur mit numerischen Operanden g.ltig. Es wird zun.chst die Exponentation durchgef.hrt, danach das Ergebnis in <idnVar> abgelegt.

Hinweis

Es ist zu beachten, da. dieser Operator ein In-Line-Operator ist und somit auch in jedem Befehl oder in jeder Funktion verwendet werden darf, deren Syntax den Einsatz eines Ausdrucks erlaubt.

Beispiele

```
nZahl := 10
? nNeueZahl := ( nZahl ^= 2 ) / 5    // Resultat:  20
? nZahl                              // Resultat: 100
```

++ Inkrementation -- un.r (mathematisch)

Syntax

++<idVar>
<idVar>++

Typ

Numerisch, Datum

Operanden

<idVar> ist der Bezeichner einer g.ltigen und numerisch oder mit Datumswert initialisierten Clipper-Variable.

Beschreibung

Der un.re Inkremental-Operator ++ kann in Verbindung mit numerischen oder Datums-Variablen benutzt werden, um den Inhalt der Variable um 1 zu erh.hen.

Hinweis

Im Gegensatz zu anderen Operatoren, die auch auf komplexe Ausdr.cke angewendet werden d.rfen, arbeitet dieser Operator nur mit einzelnen Speichervariablen (keine Datenbankfelder) zusammen.

Der Operator kann mit Prefix- oder Postfix-Notation benutzt werden. Wird er in Verbindung mit einer Zuweisung verwendet, .ndert die Prefix-Notation den Wert vor und die Postfix-Notation den Wert nach der Zuweisung.

Beispiele

```
nZahl := 10
nNeueZahl := ++nZahl              // Resultat: 11
? nZahl, nNeueZahl                // Resultat: 11 , 11
nNeueZahl := nZahl++
? nZahl, nNeueZahl                // Resultat: 12 , 11

// Die folgende Schleife:
FOR i := 1 TO 10
  ? i
NEXT

// kann auch wie folgt geschrieben werden:
i := 1
WHILE  i < 10
  ? i++
END
```

```
--         Dekrementation              -- un.r              (mathematisch)
-------------------------------------------------------------------------
```

Syntax

```
  --<idVar>
  <idVar>--
```

Typ

Numerisch, Datum

Operanden

<idVar> ist der Bezeichner einer g.ltigen und numerisch oder mit Datums-
wert initialisierten Clipper-Variable.

Beschreibung

Der un.re Dekremental-Operator -- kann in Verbindung mit numerischen
oder Datums-Variablen benutzt werden, um den Inhalt der Variable um 1
zu erniedrigen.

Hinweis

Im Gegensatz zu anderen Operatoren, die auch auf komplexe Ausdr.cke an-
gewendet werden d.rfen, arbeitet dieser Operator nur mit einzelnen
Speichervariablen (keine Datenbankfelder) zusammen.

Der Operator kann mit Prefix- oder Postfix-Notation benutzt werden. Wird
er in Verbindung mit einer Zuweisung verwendet, .ndert die Prefix-Notation
den Wert vor und die Postfix-Notation den Wert nach der Zuweisung.

Beispiele

```
  nZahl := 10
  nNeueZahl := --nZahl
  ? nZahl, nNeueZahl                // Resultat: 9 , 9
  nNeueZahl := nZahl--
  ? nZahl, nNeueZahl                // Resultat: 8 , 9
```

Syntax

&<cMacroVar>[.]
&(<cMacroVar>)

Operanden

<cMacroVar> ist eine g.ltige Clipper-Variable vom Typ Zeichen. Sie kann durch einen Punkt begrenzt werden, wenn die Textsubstitution mit weiteren Ausdr.cken kombiniert werden soll.

<cMacroVar> ist ein Zeichenkettenausdruck, der in Klammern eingeschlossen wird. Der Ausdruck wird zuerst berechnet, bevor das Ergebnis als Makro-operation ausgef.hrt wird.

Beschreibung

Der Makro-Operator in Clipper 5.0 erlaubt die Compilation von Ausdr.cken zur Laufzeit, sowie eine Textsubstitution innerhalb von Zeichenketten. Sobald der Makro-Operator zur Ausf.hrung gelangt, wird der Operand an einen speziellen Laufzeitcompiler (den Makro-Compiler) .bergeben. Dieser Makro-Compiler kann nur Ausdr.cke, keine Statements oder Befehle .bersetzen.

Textsubstitution

Immer wenn eine Referenz auf eine PRIVAT- oder PUBLIC-Variable mit einer Zeichenkette verkn.pft werden soll, wie zum Beispiel:

```
cMakro := "Ihr da"
? "Hallo , &cMakro"
```

wird der Inhalt der Makro-Variablen anstelle der Referenz in den Text eingesetzt.

Laufzeitcompilierung

Wird eine Makro-Variable oder ein Makro-Ausdruck innerhalb eines Ausdrucks angegeben, so arbeitet der Makro-Operator (&) als "compilieren und ausf.hren" Symbol. Wird ein Makro also wie folgt angesprochen:

```
cMakro := "DTOC(DATE())"
? &cMakro
```

compiliert der Makro-Compiler den Inhalt von cMakro und f.hrt diesen danach aus. Der compilierte Code wird nach der Ausf.hrung verworfen. .bergibt man einen in Klammern eingeschlossenen Ausdruck an den Makro-Operator, wie hier:

```
? &(INDEXKEY(0))
```

so wird der Ausdruck zuerst ausgef.hrt und die resultierende Zeichen-kette dann wie eine Makro-Variable compiliert und ausgef.hrt.

Hinweis

. Makros und Befehle: Der Makro-Operator kann nicht dazu benutzt werden, um Befehlsschl.sselworte zu compilieren oder zu ersetzten.

. Makros als Befehlsargumente: In fr.heren Clipper-Versionen und auch in einigen dBase-Sprachen wurden Makros h.ufig in Befehlen eingesetzt, die eine Textkonstante als Argument ben.tigen. Das betraf die Datenbank-befehle genauso wie zum Beispiel die SET-Schallter. In solchen F.llen k.nnen nun erweiterte Ausdr.cke angewendet werden. So kann zum Beispiel:

```
xcDatenbank := "Kunden"
USE &xcDatenbank
```

durch

```
xcDatenbank := "Kunden"
```

```
USE (xcDatenbank)
```

ersetzt werden. Dies ist besonders bei der Verwendung von LOCAL- und STATIC-Variablen wichtig, da diese nicht mit dem Macro-Operator substituiert werden k.nnen.

. Makros und Listen: Der Makro-Operator (&) kann keine Argumentlisten substituieren oder compilieren, wie sie in vielen Befehlen benutzt werden. Argumentenlisten sollten daher immer als erweiterte Ausdr.cke spezifiziert werden, wie das folgende Beispiel zeigt:

```
LOCAL xcIndex := { "INDEX1", "INDEX2" }
SET INDEX TO (xcIndex[1], xcIndex[2])
```

ASORT() Sortiert ein Array

Syntax

ASORT(<aArray>, [<nStart>], [<nAnzahl>], [<bOrder>]) -> aArray

Parameter

<aArray> ist das Array, dessen Elemente sortiert werden sollen.

<nStart> ist die Position des Elements, bei dem der Sortiervorgang beginnen soll. Wird der Parameter nicht angegeben, so beginnt die Sortierung beim ersten Element.

<nAnzahl> gibt die Anzahl der Elemente an, die beginnend bei <nStart> sortiert werden sollen. Wird der Parameter nicht angegeben, so wird bis zum letzten Element von <aArray> sortiert.

<bOrder> ist ein optionaler Codeblock, mit dem die Sortierfolge bestimmt werden kann. Wird der Parameter nicht angegeben, wird in aufsteigender Reihenfolge sortiert.

R.ckgabe

ASORT() gibt die Referenz des sortierten Arrays <aArray> zur.ck.

Anwendung

ASORT() ist eine Arrayfunktion, die ein gesamtes oder einen Teil eines Arrays, welches Elemente gleichen Typs enth.lt, sortiert. Es k.nnen die Datentypen Zeichen, Datum, Logisch und Numerisch sortiert werden.
Wird <bOrder> nicht angegeben, so werden die Elemente in aufsteigender Reihenfolge sortiert. Elemente, die eine geringere Wertigkeit besitzen, werden an den Anfang, und Elemente mit h.herer Wertigkeit an das Ende des Arrays gestellt.

Wird <bOrder> als Parameter angegeben, so bestimmt dieser die Sortierfolge. F.r jedes Element werden dem Codeblock jeweils zwei Arrayelemente .bergeben, bevor dieser ausgef.hrt wird. Der Block mu. immer dann true (.T.) liefern, wenn die beiden Elemente in der richtigen Folge vorliegen, andernfalls mu. er false (.F.) liefern.
Diese M.glichkeit kann dazu benutzt werden, um absteigend oder in einer andereren vorgegebenen Reihenfolge zu sortieren. Sehen Sie dazu die Beispiele.

Bei der Sortierung werden Zeichen oder Zeichenketten in der Folge ihres ASCII-Wertes, logische Werte mit false (.F.) als niederwertig, mit true (.T.) als h.herwertig, Datumswerte in chronologischer Folge und numerische Werte entsprechend ihrer Wertigkeit, sortiert.

Hinweis

. Da in Clipper mehrdimensionale Arrays durch Verschachtelung von Subarrays erzeugt werden, k.nnen Sie mit ASORT() nicht auf direktem Wege sortiert werden. Um mehrdimensionale Arrays zu sortieren, kann ein Codeblock angegeben werden, welcher die Sortierung der Subarrays vornimmt.

Beispiele

. Das Beispiel erzeugt ein Array mit f.nf unsortierten Elementen, sortiert das Array zun.chst in aufsteigender Reihenfolge und danach in absteigender Reihenfolge durch einen .bergebenen Codeblock:

```
aArray := { 3, 5, 1, 2, 4 }
ASORT(aArray)                          // Resultat: 1,2,3,4,5
ASORT(aArray,,, {  | x,y x > y })     // Resultat: 5,4,3,2,1
```

. Das Beispiel sortiert ein Array mit Zeichenketten in aufsteigender Reihenfolge unabh.ngig von Gro.- und Kleinschreibung. Dies wird durch die Angabe eines Codeblocks erreicht, welcher den Vergleich der Elemente .ber die Funktion UPPER() vornimmt.

```
aArray := {"Friedhelm", "Katrin", "Albert"}
```

```
ASORT(aArray,,, { |x,y| UPPER(x) < UPPER(y) }
```

. Das n.chste Beispiel sortiert ein mehrdimensionales Array .ber das zweite Element jedes Subarrays:

```
aKids     :=  {{"Sebastian", 10},{"Anne", 23},{"Phillip", 7}}
aSortKids   :=  ASORT(aKids,,, {|x,y| x[2] < y[2] }
// Resultat :{{"Phillip", 7},{"Sebastian", 10},{"Anne", 23}}
```

Dateien

Bibliothek ist EXTEND.LIB

See Also:

```
->         Alias-Operation             -- un.r                   (Spezial)
-------------------------------------------------------------------------
```

Syntax

```
  <idAlias>-><idField>
  <idAlias>->(<Exp>)
  FIELD-><idVar>
  MEMVAR-><idVar>
```

Operanden

<idAlias> ist der Bezeichner eines Arbeitsbereiches, in dem eine Datei ge.ffnet ist.

<idField> ist der Feld-Bezeichner einer im Arbeitsbereich <idAlias> ge.ffneten Datei.

(<Exp>) ist ein Ausdruck beliebigen Datentyps, der sich auf den mit <idAlias> bezeichneten Arbeitsbereich bezieht. Der Ausdruck mu. in Klammern eingeschlossen sein.

<idVar> ist der Bezeichner einer Clipper-Variablen. Abh.ngig davon, ob FIELD oder MEMVAR spezifiziert ist, wird eine Feldvariable oder eine Speichervariable deklariert.

Beschreibung

Wenn vor dem Alias-Operator (->) ein Aliasbezeichner als Operand benutzt wird, bezieht sich der Feldname (<idField>) oder der Ausduck (<Exp>) auf den, mit dem Aliasbezeichner angegebenen Arbeitsbereich. Der Operator ist nur g.ltig, wenn im angegebenen Arbeitsbereich eine Datenbankdatei ge.ffnet wurde. Der Operator erm.glicht den Zugriff auf Informationen aus Arbeitsbereichen, die nicht explizit selektiert sind.

Beispiele

@ .bergabe per Referenz -- un.r (Spezial)

Syntax

@<idVar>

Operanden

<idVar> ist ein Bezeichner einer g.ltigen Clipper-Variablen, die als Referenz .bergeben werden soll.

Beschreibung

.bergabe als Referenz hei.t, da. ein Verweis auf den Wert des Argumentes .bergeben wird und nicht eine Kopie der Daten. Der Parameter zeigt dann auf die gleiche Speicherstelle wie das Argument. Falls die aufgerufene Routine den Wert des Parameters .ndert, wird auch der Wert des Argumentes im aufrufenden Programm ge.ndert.

Wird die Syntax f.r Funktionsaufrufe eingesetzt, k.nnen alle Variablen bis auf Feldvariablen per Referenz .bergeben werden, indem der @-Operator dem entsprechenden Argument vorangestellt wird:

```
nX := nY := 1
Increment( @nX, nY )
? nX, nY                        // Resultat: 2, 1

PROCEDURE Increment( nZahl1, nZahl2 )
  ++nZahl1
  ++nZahl2
RETURN
```

In diesem Beispiel werden innerhalb der Prozedur Increment beide .bergebenen Parameter um 1 erh.ht. Da das erste Argument als Referenz und das zweite Argument als Wert (per value) .bergeben wird, .ndert sich nach Verlassen der Prozedur lediglich das erste Argument, da die Kopie des Wertes des zweiten Arguments wieder verworfen wird.

{ } Arraykonstanten- oder Codeblock-Begrenzungszeichen (Spezial)

Syntax

{<Konst1> [,<Konstn>...]}
{|[<idPara1> [,<idParan>...]]| <Exp> }

Operanden

<Konst> sind Konstanten, die Werte der Elemente innerhalb des literal definierten Array beschreiben.

<idPara> sind Bezeichner f.r Parameter, die als Argumente dem definierten Code-Block .bergeben werden.

<Exp> ist ein g.ltiger Ausdruck, der bei Ausf.hrung des Code-Blockes berechnet und als Ergebnis zur.ckgegeben wird.

Beschreibung

Geschweifte Klammern werden in Clipper zur Definition literaler Arrays und Code-Bl.cke verwendet.

Bei der Definition literaler Arrays werden die Werte der einzelnen Elemente innerhalb der geschweiften Klammern durch Komma getrennt angegeben:

```
aArray := { 10, 20, 30, 40 }
```

Bei der Definition von Code-Bl.cken m.ssen innerhalb der geschweiften Klammern zwei senkrechte Balken (|) angegeben werden. Innerhalb der Balken werden optional g.ltige Bezeichner f.r eventuell dem Code-Block .bergebene Parameter angegeben:

```
bBlock := { | x | QOUT(x) }
```

Beispiele

. Im folgenden Beispiel wird ein zweidimensionales Array literal definiert:

```
aArray := { {10, 20}, { 10, 30}, { 10, 40} }
```

. Das n.chste Beispiel definiert einen Code-Block, der das Maximum zweier Argumente berechnet:

```
bMax := {| x, y| IF( x > y, x, y)}
```

[ ] Array-Element Indikator (Spezial)

Syntax

<aExp>[nID,...]
<aExp>[nID][nID]...

Beschreibung

Der Array-Elemente-Indikator ([]) wird ben.tigt, um die Werte einzelner Elemente eines Arrays ansprechen zu k.nnen. Der Bezeichner des Arrays mu. den Indikatoren vorangestellt und die Indikatoren innerhalb der eckigen Klammern gestellt werden.

Beispiele

Das folgende Beispiel zeigt den Zugriff auf ein zweidimensionales Array:

```
FOR i := 1 TO LEN(aArray)
  FOR j := 1 TO LEN(aArray[i])
    ? aArray[i,j]
  NEXT
NEXT
```

```
( )        Funktions- oder Gruppierungs-Indikator                (Spezial)
-------------------------------------------------------------------------
```

Syntax

Typ

Operanden

Beschreibung

Hinweis

Beispiele

SET CLIPPER Laufzeit-Konfiguration f.r Clipper-compilierte Programme

---

SET CLIPPER=[E<nExpandedKBytes>]
[;F<nHandles>]
[;X<nKBytes>]
[;DYNK:<nKBytes>]
[;DYNF:<nHandles>]
[;SWAPK:<nKBytes>]
[;SWAPPATH:"<path>"]
[;CGACURS]

Beschreibung

Clipper unterst.tzt die M.glichkeit, das Verhalten eines mit Clipper erstellten Programmes .ber eine DOS-Umgebungsvariable CLIPPER zu .ndern. Wenn ein Clipper-Programm aufgerufen wird, sucht es nach der Umgebungsvariablen CLIPPER und reserviert entsprechenden Speicherplatz oder Datei-Handles, so wie sie als Parameter in der DOS-Variablen CLIPPER angegeben sind. Die DOS-Variable Clipper kann von der DOS-Kommandozeile eingegeben, oder in eine BATCH-Datei (z.B. AUTOEXEC.BAT) eingegeben werden. Soll der Benutzer ein Clipper-Programm in einem Netzwerk starten, so kann die DOS-Variable auch in einem LOGIN-SCRIPT angegeben werden.

Anwendung

. **E<nExpandedKBytes>** begrenzt den zur Verf.gung stehenden Expanded-Memory-Bereich, auf <nExpandedKBytes>. Die mit Clipper erstellten Programme k.nnen auf Expanded-Memory zugreifen, um die Ausf.hrungsgeschwindigkeit zu erh.hen. Clipper kann auf erweiterten Speicher zugreifen, wenn er als LOTUS-INTEL-MICROSOFT (LIM) Expanded-Memory-Spezifikation (EMS) Version 3.2 oder h.her eingerichtet ist.

  Expanded-Memory wird automatisch bei Programmstart durch ein aufgerufenes Clipper-Programm reserviert; es erfolgt keine dynamische Speicherplatz-reservierung w.hrend der Ausf.hrung eines Programmes. Der maximal f.r ein Clipper-Programm zur Verf.gung stehende Speicher wird durch vier Faktoren bestimmt:

  . Durch die EMS-Speichergr..e
  . Durch die .ber SET CLIPPER=Ennn erfolgte Begrenzung
  . Durch ein theoretisches Maximum von 8MB
  . Durch den verf.gbaren konventionellen Speicherbereich

  Beachten Sie, da. ein Teil des konventionellen Speicherbereiches f.r die virtuelle Speicherverwaltung als Speichertabelle ben.tigt wird.

. **F<nHandles>** wird in Verbindung mit der Files-Einstellung in der CONFIG.SYS benutzt, um die maximale Anzahl der gleichzeitig offenen Dateien in einem Clipper-Programm zu bestimmen. Die Einstellung wird nur dann ben.tigt, wenn mehr als 20 gleichzeitig ge.ffnete Dateien von einem Clipper-Programm ben.tigt werden und das Programm auf DOS-Version 3.3 oder h.her eingesetzt wird. Die Anzahl der m.glichen, gleichzeitig ge.ffneten Dateien wird bestimmt durch die kleinste der beiden Angaben (FILES, in CONFIG.SYS und Fnn-Parameter in der DOS-Variable CLIPPER). Ist zum Beispiel die Einstellung FILES auf 120 und der F-Parameter auf 50 gesetzt, lassen sich in einem Clipper-Programm max. 50 Dateien gleich-zeitig .ffnen. Die ideale F-Einstellung ist eine ungerade Zahl, die um 5 h.her ist als die FILES-Einstellung.

. **X<nKBytes>** definiert eine Gr..e oder Speicher, die von der Speicher-reservierung ausgenommen werden soll (mit Ausnahme der RUN Befehle). Der Wert wird in KByte angegeben und liegt im Bereich zwischen 0 und 64. Wird der Parameter nicht angegeben, wird der gesamte Speicherbereich reserviert.

. **DYNK:<nKBytes>** kontrolliert den Speicherbereich der f.r dynamische Overlays bestimmt ist. Der Wert wird in kByte im Bereich 4 bis 64 (inkl) angegeben. Wird der Parameter nicht angegeben, wird automatisch der g.nstigste Wert ermittelt, was zu empfehlen ist.

. **DYNF:<nHandles>** spezifiziert die Anzahl der Datei-Handles, die das dynamische Overlaysystem verwenden darf. G.ltige Angaben liegen zwischen 1 und 8. Nicht angegeben wird dieser Wert automatisch auf 2 gesetzt.

. **SWAPK:<nKBytes>** gibt die maximale Gr..e der tempor.ren Auslagerungs-Datei (Swap file) f.r das virtuelle Memory-System (VMM) an. G.ltige Angaben liegen zwischen 256 und 16535. Der voreingestellte Wert ist 16384 (16MB).

. **SWAPPATH:"<path>"** spezifiziert das Laufwerk und/oder das Verzeichnis, wo die temor.re Auslagerungsdatei (swap file) f.r fas virtuelle Memory-System (VMM) plaziert werden soll. Die Voreinstellung ist das aktuelle Verzeichnis.

Hinweis
Beachten Sie, da. die tempor.re Datei f.r das virtuelle Memory-System (VMM) immer nur so gro. wie ben.tigt wird; die Angabe von SWAPK bedeutet nicht, da. die Datei auch tats.chlich so gro. wird.

. **CGACURS** gibt an, da. das Programm die erweiterten M.glichkeiten der Cursor-Darstellung von EGA/VGA benutzt. Diese Angabe bewirkt, da. erweiterte Cursor-Modi dieser Bildschirmadapter unterst.tzt werden.

Beispiel

```
SET CLIPPER=F50;DYNF:4;DYNK:4;SWAPK:4096;SWAPPATH:"C:\CLP\SWAP"
```

SET CLIPPERCMD Voreinstellen der Compileroptionen

SET CLIPPERCMD=<CompilerOptionsListe>

Argument

<CompilerOptionsListe> ist eine Liste mit Compilerschaltern, die bei jedem Aufruf des Clipper-Compilers benutzt wird.

Beschreibung

Die DOS-Environment-Variable CLIPPERCMD wird vom Compiler zur Voreinstellung von Compiler-Optionen verwendet. Diese Angaben werden .berschrieben, wenn die gleichen Optionen in der Compiler-Kommandozeile angegeben werden.

Beispiel

```
SET CLIPPERCMD=/M/W/B
```

SET INCLUDE Verzeichnis zur Suche von Header-Dateien (compiler)

SET INCLUDE=<VerzeichnisListe>

Argument

<VerzeichnisListe> ist eine oder sind mehrere Verzeichnisse, die durch Semikolon (;) getrennt angegeben werden.

Beschreibung

Die DOS-Environment-Variable INCLUDE wird vom Compiler benutzt, um nach Header-Dateien zu suchen, die sich nicht im aktuellen Verzeichnis befinden. Wird die Compiler-Option /I benutzt, um ein weiteres Verzeichnis f.r Header-Dateien anzugeben, wird diese Verzeichnis zuerst durchsucht, bevor das aktuelle Verzeichnis und anschlie.end das INCLUDE-Verzeichnis durchsucht wird.

Das durch die Installation vorgegebene Verzeichnis ist \CLIPPER5\INCLUDE.

Beispiel

```
SET INCLUDE=C:\CLIPPER5\INCLUDE
```

SET LIB Verzeichnis zur Suche von .LIB,.PLT,.PLL (linker,runtime)

SET LIB=<VerzeichnisListe>

Argument

<VerzeichnisListe> ist eine oder sind mehrere Verzeichnisse, die durch Semikolon (;) getrennt angegeben werden.

Beschreibung

Die DOS-Environment-Variable LIB wird von RTLink benutzt, um nach Bibliotheks- (.LIB), PRE-LINK-Transfer- (.PLT) und PRE-LINK-LIBRARY-Dateien zu suchen, die sich nicht im aktuellen Verzeichnis befinden.
RTlink sucht zuerst im Verzeichnis, das .ber PLL spezifiziert wurde, bevor das LIB-Verzeichnis durchsucht wird. Zur Laufzeit wird das LIB-Verzeichnis benutzt, um nach der PLL-Datei zu suchen, wenn sich diese Datei nicht im aktuellen oder PLL-Verzeichnis befindet.

Das durch die Installation vorgegebene Verzeichnis ist \CLIPPER5\LIB.

Beispiel

```
SET LIB=C:\CLIPPER5\LIB
```

SET OBJ Verzeichnis zur Suche von .OBJ-Dateien (linker)

SET OBJ=<VerzeichnisListe>

Argument

<VerzeichnisListe> ist eine oder sind mehrere Verzeichnisse, die durch Semikolon (;) getrennt angegeben werden.

Beschreibung

Die DOS-Environment-Variable OBJ wird von RTLink benutzt, um nach Objekt-dateien (.OBJ) zu suchen, die sich nicht im aktuellen Verzeichnis befinden.

Objekt-Dateien, die von Clipper erzeugt werden, legt dieser im aktuellen Verzeichnis ab, wenn nicht die /O Option ein anderes Verzeichnis verlangt. Durch Setzen der DOS-Variablen CLIPPERCMD:

```
SET CLIPPERCMD=/OC:\CLIPPER5\OBJ\
```

k.nnen alle Objekdateien in einem speziellen Verzeichnis angelegt werden, auf das beim Linken durch die Einstellung:

```
SET OBJ=C:\CLIPPER5\OBJ
```

zugegriffen werden kann.

Beispiel

```
SET OBJ=C:\CLIPPER5\LIB
```

SET PLL Verzeichnis zur Suche von .PLT und .PLL (linker,runtime)

SET PLL=<VerzeichnisListe>

Argument

<VerzeichnisListe> ist eine oder sind mehrere Verzeichnisse, die durch Semikolon (;) getrennt angegeben werden.

Beschreibung

Die DOS-Environment-Variable PLL wird von RTLink benutzt, um nach .PLL- und .PLT-Dateien zu suchen. Diese Dateien werden nicht im aktuellen Verzeichnis gesucht.

Die PLL-Verzeichnisangabe wird auch von den fertigen Clipper-Programmen benutzt, um nach .PLL-Dateien zu suchen.

Das durch die Installation vorgegebene Verzeichnis ist \CLIPPER5\PLL.

Beispiel

```
SET PLL=C:\CLIPPER5\PLL
```

SET RMAKE Voreinstellung der RMAKE-Optionen (rmake)

SET RMAKE=<RMakeOptionsListe>

Argument

<RMakeOptionsListe> ist eine Liste mit Optionen, die bei jedem Aufruf von RMake benutzt wird.

Beschreibung

Die DOS-Environment-Variable RMAKE wird von RMake zur Voreinstellung von Optionen verwendet. Diese Angaben werden .berschrieben, wenn die gleichen Optionen in der RMake-Kommandozeile angegeben werden.

Beispiel

```
SET RMAKE=/B/W
```

SET RTLINKCMD Voreinstellung der Linker-Optionen (linker)

SET RTLINKCMD=<LinkOptionsListe>

Argument

<LinkOptionsListe> ist eine Liste mit Optionen, die bei jedem Aufruf des RTLink benutzt wird.

Beschreibung

Die DOS-Environment-Variable RTLINKCMD wird von RTlink zur Voreinstellung von Optionen verwendet. Diese Angaben werden .berschrieben, wenn die gleichen Optionen in der RTLink-Kommandozeile angegeben werden.

Beispiel

```
SET RTLINKCMD=/POSITIONAL /INCR:50
```

SET TMP Verzeichnis f.r tempor.re Dateien (compiler,linker)

SET TMP=<Verzeichnis>

Argument

<Verzeichnis> ist ein Verzeichnis, in dem tempor.re Dateien abgelegt werden.

Beschreibung

Die DOS-Environment-Variable TMP spezifiziert ein Verzeichnis, in das vom Clipper Compiler und von RTLink tempor.re Arbeits-Dateien abgelegt werden. Wird diese DOS-Variable nicht spezifiziert, werden alle tempor.ren Dateien im aktuellen Verzeichnis abgelegt.
Die Compiler-Option /T kann diese Verzeichnisangabe f.r den aktuellen Compiliervorgang .berschreiben.

Bei Compilier- und Link-Vorg.ngen werden f.r verschiedene Prozesse tempor.-re Arbeitsdateien gebildet. Diese Dateien existieren nur f.r den aktuellen Vorgang und werden bei Beendigung des Compilier- oder Link-Vorganges wieder gel.scht.

Hinweis

Im Gegensatz zu Clipper 5.0 wurden bei Clipper S87 tempor.re Dateien in das Verzeichnis, welches .ber SET DEFAULT angegeben war, oder in das aktuelle DOS-Verzeichnis abgelegt, falls SET DEFAULT nicht angegeben war.

Beispiel

```
SET TMP=C:\TEMP
```

Die .nderung der Environment-Gr..e

Wenn die Umgebungsvariablen CLIPPERCMD, INCLUDE und TMP angegeben werden, kann es vorkommen, da. die Gr..e des DOS-Umgebungsspeichers (Environment) pl.tzlich nicht mehr ausreicht. Ab der DOS-Version 3.2 k.nnen Sie diese Gr..e jedoch anpassen, indem Sie in der Datei CONFIG.SYS den Kommando-Prozessor COMMAND.COM in Verbindung mit der SHELL-Anweisung laden.

Das folgende Beispiel l.dt COMMAND.COM mit einer Umgebungsspeicher-Gr..e von 2.048 Bytes:

```
SHELL=C:\COMMAND.COM C:\ /P /E:2048
```

Neben der Angabe der Umgebungsgr..e durch /E m.ssen Sie auch Namen und Pfad von COMMAND.COM sowie die Option /P angeben, damit COMMAND.COM resident im Speicher verbleibt.

Der Aufruf des Clipper-Compilers

Der Clipper-Compiler kann vom DOS-Prompt aus mit der folgenden allgemeinen Syntax aufgerufen werden:

CLIPPER [<QuellDatei> | @<ScriptDatei>] [<OptionenListe>]

<QuellDatei> ist der Name der Programmdatei, die in eine Objekt-Datei compiliert werden soll. Falls keine andere Erweiterung angegeben ist, so wird die Standard-Erweiterung .PRG angenommen. Der Dateiname kann zus.tzlich einen Laufwerksbezeichner und einen Pfadnamen enthalten.

<ScriptDatei> ist der Name einer Datei, welche eine Liste derjenigen Quelldateien enth.lt, die zusammen in eine einzige Objekt-Datei compiliert werden sollen. Falls keine andere Erweiterung angegeben ist, so wird die Standard-Erweiterung .CLP angenommen. Die Script-Datei wird gelegentlich auch als Clip-list bezeichnet.

<OptionenListe> ist eine Liste mit einer oder mehreren Optionen, die das Ergebnis der Compilierung beeinflussen. Eine Option kann als Klein- oder Gro.buchstabe angegeben werden, wobei diesem ein Schr.g- oder Bindestrich vorausgehen mu.. Die einzelnen Compiler-Optionen werden in diesem Kapitel im Abschnitt Compiler-Optionen beschrieben. Wenn Sie den Befehl CLIPPER ohne Angabe von weiteren Parametern ausf.hren, wird eine Liste und Kurzbeschreibung der m.glichen Optionen ausgegeben.

Beispiele

. Die folgende Befehlszeile bewirkt, da. der Compiler nur die Syntax und nur in Haupt.prg .berpr.ft und keine Zeilennummern anzeigt:

```
CLIPPER Haupt /M /Q /S
```

. Im folgenden Beispiel werden alle in der Script-Datei Auftrag.clp aufgelisteten Dateien compiliert. Die angegebenen Optionen haben zur Folge, da. die Debugging-Informationen in die Objekt-Datei mit eingeschlossen werden, die unter dem Namen Auftrag1.obj im Verzeichnis C:\CLIPPER5\OBJ abgelegt wird.

```
CLIPPER @Auftrag /B /OC:\CLIPPER\5\OBJ\Auftrag1
```

Die Script-Datei k.nnte dabei zum Beispiel folgende Dateien enthalten:

```
AufHaupt
AufKunde
AufRech
AufStat
```

See Also:

Compiler-Optionen

Compiler-Optionen stellen Schalter dar, die das Ergebnis des Compilierungs-Prozesses beeinflussen und steuern. Neben den Pre-Prozessor-Direktiven und der Standard-Header-Datei sind Compiler-Optionen die Hauptmechanismen zur Kontrolle des Compilers und Pre-Prozessors. Die Optionen k.nnen in der Clipper-Befehlszeile und .ber die Umgebungsvariable CLIPPERCMD angegeben werden. Bei Konflikten durch doppelte Angaben hat immer die Option in der Clipper-Befehlszeile Vorrang.

Es existieren drei Kategorien von Compiler-Optionen:

. Variablen-Konfiguration

. Pre-Prozessor-Schalter

. Compiler-Schalter

Optionen k.nnen durch Klein- oder Gro.buchstaben angegeben werden, wobei jeder Option ein Schr.g- (/) oder Bindestrich (-) vorausgehen mu.. Die Optionen k.nnen in beliebiger Reihenfolge aufgef.hrt werden, m.ssen aber durch ein Leerzeichen voneinander getrennt sein. Einige Compiler-Optionen haben zus.tzlich einen Parameter. Die Syntax dieser Optionen ist im folgenden jeweils in der nach der Options.berschrift folgenden Zeile angegeben. Zwischen der Option und dem zugeh.rigen Parameter darf kein Leerzeichen stehen.

/A Automatische Deklaration von PRIVATE- und PUBLIC-Variablen

---

Jede in einer PRIVATE-, PUBLIC- oder PARAMETERS-Anweisung aufgef.hrte Variable wird automatisch als MEMVAR deklariert.

/B Einbeziehung der Debugging-Informationen

Bei Angabe dieser Option werden in die Objekt-Datei Informationen f.r den Debugger aufgenommen, z.B. Zeilennummern sowie Namen von LOCAL- und STATIC-Variablen und die der Quelldateien. Wenn die Option /L angegeben worden ist, werden die Zeilennummern nicht als Teil der Debugging-Informationen mit aufgenommen.

/D Definition eines Bezeichners

/D<Bezeichner>[=<Text>]

Definiert einen Bezeichner f.r den Pre-Prozessor. Wenn <Text> nicht angegeben ist, wird <Bezeichner> ein leerer Wert ("") zugewiesen. Falls Sie ein Anf.hrungszeichen als Literalzeichen in <Text> aufnehmen m.chten, mu. diesem Anf.hrungszeichen ein Backslash (\) vorangestellt werden.

Diese Option wird in Verbindung mit den bedingten Direktiven des Pre-Prozessors zur Compilierung verwendet. Durch Definition einer symbolischen Konstanten auf der Compiler-Befehlszeile kann ein Teil der Quelldatei von der Compilierung ausgeschlossen oder umgekehrt miteingeschlossen werden. Ausf.hrliche Informationen zum Einsatz dieser Direktiven erhalten Sie im Kapitel Direktiven des Referenz-Handbuches I.

/I Erweiterung der Suchpfade f.r Include-Dateien

/I<PfadName>

Stellt der mit der Umgebungsvariablen INCLUDE spezifizierten Suchpfadliste f.r INCLUDE-Dateien den zus.tzlichen Suchpfad <PfadName> voran. Durch mehrfache Angabe der Option /I k.nnen entsprechend viele Suchpfade zur bereits vorhandenen INCLUDE-Pfad- liste hinzugef.gt werden.

/L Unterdr.ckung von Zeilennummern

Nimmt die Zeilennummern des Quellcodes nicht mit in die erzeugte Objekt-Datei auf. Dadurch wird die Objekt-Datei pro Zeile, die eine Programmanweisung enth.lt, um 3 Bytes verkleinert. Diese Option sollte aber nur bei komplett fertiggestellten und getesteten Applikationen verwendet werden, da sonst die Ausgabe der fehlerhaften Zeile bei einem runtime error entf.llt. Au.erdem kann ein mit der Option /L compiliertes Programm nicht mit dem Clipper Debugger bearbeitet werden. Wenn sowohl /B als auch /L beim Compilieren angegeben sind, dann werden die Zeilennummern nicht als Debugging-Informationen in die resultierende Objekt-Datei eingeschlossen.

/M Compilierung nur der angegebenen Programmdatei

Bei Angabe dieser Option wird nur die in der Befehlszeile angegebene .PRG-Datei compiliert. Der Compiler sucht nicht nach unresolved externals, d.h. nach in der Programmdatei aufgerufenen Funktionen oder Prozeduren, die nicht in der augenblicklich compilierten .PRG-Datei enthalten sind. Wenn /M verwendet wird, erfolgt keine automatische Suche und Compilierung von Prozeduren, auf die mit DO, SET FORMAT oder SET PROCEDURE verwiesen wird und die nicht in der aktuellen .PRG-Datei gefunden wurden. Mit SET PROCEDURE aktivierte Prozedurdateien werden nicht compiliert. Header-Dateien, die mit der Direktive #include aufgef.hrt sind, werden hingegen compiliert.

/M beeinflu.t die Compilierung mit Script-Dateien nicht.

/N Unterdr.ckung der automatischen Hauptprozedur

Bei Angabe von /N wird keine automatische Definition einer Prozedur vorgenommen, die den gleichen Namen wie die augenblicklich compilierte Programmdatei erh.lt. Die Option /N mu. dann eingesetzt werden, wenn Sie in einer .PRG-Datei Variablendeklarationen benutzt werden, welche f.r die komplette Programmdatei gelten sollen.

Wenn die Option /N nicht angegeben ist, dann wird jede .PRG-Datei in der Weise compiliert, da. eine implizite Prozedur mit dem Namen der angegebenen .PRG-Datei definiert wird, die den Code vom Dateianfang bis zur ersten expliziten PROCEDURE- oder FUNCTION-Deklaration enth.lt. Die nach Aufruf einer .EXE-Datei zuerst ausgef.hrte Prozedur (Startprozedur) ist immer die erste Prozedur oder Funktion, die beim Linken entdeckt wird. Bei Compilierung ohne die Option /N wird die eben beschriebene implizit erzeugte Prozedur zur Startprozedur. Bei Compilierung mit der Option /N wird die erste explizit deklarierte Prozedur oder Funktion der ersten beim Linken aufgef.hrten Objekt-Datei zur Startprozedur.

/O Compilierung in die angegebene Objektdatei

/O<ObjektDatei>

Erzeugt eine Objekt-Datei mit dem Namen von <ObjektDatei>. Wenn <ObjektDatei> nur einen Pfadnamen enth.lt, so mu. dieser mit einem Backslash (\) abgeschlossen werden. Die erzeugte Objekt-Datei wird dann in diesen Pfad geschrieben. Sie erh.lt den Namen der ersten angegebenen Programm- bzw. Script-Datei, jedoch mit der Dateinamenserweitung .OBJ.

/P Erzeugung eines Ausgabe-Listings des Pre-Prozessors

Bei Angabe der Option /P wird ein Ausgabe-Listing des Pre-Prozessors erzeugt. Das Listing wird in eine Datei geschrieben, die den Namen der ersten angegebenen Quelldatei und die Erweiterung .PPO erh.lt. Dabei wird als Verzeichnis das der namensgebenden Datei verwendet.

/Q Unterdr.ckung der Ausgabe von Zeilennummern

Bei Angabe von /Q werden die bei der Compilierung aktuell verarbeiteten Zeilennummern nicht auf dem Bildschirm ausgegeben (Quiet-Mode) Die Zeilennummern werden aber in die Objekt-Datei mit aufgenommen, soweit nicht die Option /L verwendet worden ist.

/R Gibt dem Linker eine Bibliothek zur Suche von unresolved externals an

---

/R[<LibDatei>]

Durch Angabe von /R wird in die erzeugte Objekt-Datei eine Anweisung f.r den Linker eingef.gt, nicht aufgel.ste externe Symbole (unresolved externals) in der spezifizierten Bibliothek (.LIB-Datei) zu suchen. Bei mehrfacher Angabe der Option /R wird in jeder angegebenen <LibDatei> nach unresolved externals gesucht. Wenn Sie /R ohne den Parameter <LibDatei> angeben, wird keine Suchanweisung in die Objekt-Datei eingef.gt.

Clipper 5.0 f.gt automatisch eine Suchanweisung f.r die Dateien CLIPPER.LIB, EXTEND.LIB und DBFNTX.LIB in jede Objekt-Datei ein. Die Angabe von /R .berschreibt diese vorgegebenen Suchanweisungen.

/S .berpr.fung der Syntax

Es wird nur die Syntax der angegebenen .PRG-Datei sowie aller daraus aufgerufenen Quelldateien .berpr.ft, eine Objekt-Datei wird nicht erzeugt. Die Option /M kann verwendet werden, um die Syntax der aktuellen .PRG-Datei nur zu .berpr.fen.

/T Verzeichnis f.r tempor.re Dateien

/T<PfadName>

Diese Option bestimmt den Namen eines Pfades, in den die bei der Compilierung generierten tempor.ren Dateien geschrieben werden sollen. Die Angabe dieser Option .berschreibt den mit der Umgebungsvariablen TMP spezifizierten Pfadnamen.

/U Ausf.hrung des Pre-Prozessors mit einer anwenderdefinierten Standard-Header-Datei

/U<Anwender-Standard-Header-Datei>

Gibt den Namen einer Standard-Header-Datei an, die vom Pre-Prozessor zur .bersetzung von Standard-Befehlen verwendet wird. Normalerweise erfolgt die .bersetzung der Standard-Befehle aus dem, im Pre-Prozessor selbst definierten Befehlssatz. Die Angabe von /U ohne den Dateinamen einer Header-Datei schaltet die Standard-.bersetzung von Befehlen aus. Durch die zus.tzliche Angabe einer Header-Datei wird auschlie.lich der, in dieser Datei definierte Befehlssatz verwendet.

Die Standard-Befehlsdefinition von Clipper ist zus.tzlich in der Datei STD.CH definiert, die sich im Verzeichnis \CLIPPER5\INCLUDE befindet. Die Verwendung der im Pre-Prozessor integrierten Befehlsdefinition ist gegen.ber der Verwendung der STD.CH vorzuziehen, da dadurch die Geschwindigekt des Compiliervorgangs erh.ht und ein fester Befehlssatz garantiert wird.

Die folgenden Beispiele zeigen die unterschiedliche Verwendung der /U-Option bei Aufruf des Compilers aus der DOS-Kommandozeile:

```
C:>CLIPPER Myprog             // Compiliere mit dem Standardbefehlssatz
C:>CLIPPER Myprog /USTD.CH    // Compiliere mit STD.CH
C:>CLIPPER Myprog /UOTHER.CH  // Compiliere mit OTHER.CH
C:>CLIPPER Myprog /U          // Compiliere ohne Befehlssatz
```

Da die vom Anwender angegebene Header-Datei ein Ersatz f.r den Standard-Befehlssatz darstellt, darf die Datei lediglich Direktiven f.r den Pre-Prozessor enthalten.

Clipper f.hrt die Suche nach dieser Header-Datei wie nach allen anderen Header-Dateien aus: Zuerst im aktuellen Verzeichnis, danach in dem durch die Option /I angegebenen Pfad und schlie.lich in allen durch INCLUDE spezifizierten Pfaden.

/V Behandlung aller unklaren Variablenverweise als dynamische Variablen

---

Durch die Angabe von /V wird der Compiler angewiesen, die Variablen bei Bezugnahmen auf nicht deklarierte oder nicht mit einem Alias-Verweis angesprochene Variablennamen als PRIVATE- oder PUBLIC-Variablen anzusehen. Die Vorgabe ist die Behandlung aller unklaren Variablenverweise als Datenbankfelder (FIELD). Die Option /V bewirkt dasselbe wie die Verwendung des Alias-Verweises f.r dynamische Variablen (MEMVAR->).

/W Erzeugung von Warnungen f.r unklare Variablenverweise

Durch Angabe dieser Option wird der Compiler angewiesen, bei Bezugnahmen auf nicht deklarierte oder nicht mit einem Alias-Verweis angesprochene Variablennamen eine Warnung auszugeben. Das ist dann besonders n.tzlich, wenn Sie PRIVATE- und PUBLIC-Variablen in LOCAL- oder STATIC-Variablen konvertieren m.chten.

Die Vorgabe von Optionen mit CLIPPERCMD

Der Clipper-Compiler verwendet die Umgebungsvariable CLIPPERCMD, um die Vorgabe von Optionen zu erm.glichen, die bei jedem Compiler-Aufruf ber.cksichtigt werden sollen, ohne diese immer wieder in der Befehlszeile mit angeben zu m.ssen. Die Definition von CLIPPERCMD erfolgt mit dem DOS SET-Befehl:

SET CLIPPERCMD=[<OptionenListe>]

<OptionenListe> ist eine Liste der Compiler-Optionen, die bei jedem Aufruf des Clipper-Compilers ber.cksichtigt werden.

Die mit CLIPPERCMD definierten Optionen werden genauso angegeben, als wenn sie in der Befehlszeile mit aufgef.hrt w.rden: Jede Option wird mit einem Schr.gstrich (/) oder Bindestrich (-), gefolgt von einem Klein- oder Gro.buchstaben spezifiziert. Die mit CLIPPERCMD vorgegebenen Optionen werden allerdings vor den in der Befehlszeile angegebenen Optionen ausgef.hrt. Wenn eine Option sowohl mit CLIPPERCMD als auch in der Befehlszeile definiert worden ist, dann hat die Option aus der Befehlszeile Vorrang und .berschreibt die Definition von CLIPPERCMD.

Der SET CLIPPERCMD-Befehl kann in die DOS-Datei AUTOEXEC.BAT mit aufgenommen werden. Auf diese Weise wird eine wiederholte Befehlseingabe nach jedem Computerstart vermieden, da diese Datei automatisch nach jedem Start ausgef.hrt wird.

Beispiele

. Das folgende Beispiel bewirkt die Unterdr.ckung von Zeilennummern und die Verwendung einer neuen Header-Datei als Standardvorgabe bei jedem Aufruf von Clipper:

  SET CLIPPERCMD=/L /UNeuStd.ch

. In diesem Beispiel wird das Verzeichnis angegeben, in dem die erzeugten Objekt-Dateien abgelegt werden sollen:

  SET CLIPPERCMD=/OC:\CLIPPER5\OBJ\

Die Script-Datei des Compilers

Eine Script-Datei ist eine Textdatei, die eine Liste von Dateien enth.lt, welche gemeinsam in eine einzige Objekt-Datei compiliert werden sollen. Die erzeugte Objekt-Datei tr.gt denselben Namen wie die Script-Datei, es sei denn, die Option /O wurde mit angegeben.

In der Datei-Liste k.nnen Programm-, Prozedur- und Formatdateien enthalten sein. Nicht explizit aufgef.hrte Dateien, auf die aus einer der aufgelisteten Dateien zugegriffen wird (z.B mit SET PROCEDURE oder DO), werden nicht automatisch mit compiliert. Dies erfolgt unabh.ngig von der Angabe der Option /M. Diese Option schaltet bei der direkten Angabe einer Quelldatei in der Befehlszeile die automatische Compilierung von allen in der Quelldatei aufgerufenen Dateien ein bzw. aus.

Folgende Regeln gelten f.r eine Script-Datei:

. Die Quelldateien m.ssen durch ein Carriage Return/Line Feed voneinander getrennt werden.

. Die Quelldateien k.nnen mit oder ohne Dateinamen-Erweiterung angegeben werden. Bei Angabe ohne Erweiterung wird automatisch .PRG als Erweiterung angenommen.

. Jeder Dateiname kann zus.tzlich einen Laufwerksbezeichner und einen Pfadnamen enthalten.

Der R.ckgabecode des Compilers

Bei Auftreten eines schweren Fehlers (fatal error) oder Bet.tigen von [Strg]+[C] durch den Anwender beendet der Clipper-Compiler seine Arbeit mit dem DOS-R.ckgabecode 1. Wird der Compilierungs-Proze. normal beendet, dann ist der DOS-R.ckgabecode 0. Ein normales Ende liegt auch dann vor, wenn w.hrend der Compilierung Warnungen ausgegeben worden sind. Der R.ckgabecode kann in einer Batch-Datei mit dem DOS-Schl.sselwort ERRORLEVEL abgefragt werden.

Wird ein Fehler entdeckt, so gibt der Compiler eine entsprechende Fehlermeldung aus und es wird keine Objekt-Datei erzeugt. Eine Liste der Fehlermeldungen und deren Bedeutung finden Sie im Anhang Clipper Fehlermeldungen dieses Handbuches. Sie k.nnen diese Compiler-Meldungen mit Hilfe der DOS-Umlenkung auch in eine Datei umleiten, z.B.:

```
C>CLIPPER <QuellDatei> > Meldung.txt
```

Ein Beispiel f.r die Verwendung von ERRORLEVEL zur .berpr.fung des Compilierungs-Ergebnisses finden Sie im Abschnitt Die Batch-Datei zum Compilieren und Linken in diesem Kapitel.

Die Arbeitsweise des Clipper-Pre-Prozessors

Clipper 5.0 besteht aus zwei unterschiedlichen Teilen: dem Pre-Prozessor und dem Compiler. Diese beiden Phasen werden nach Aufruf von Clipper nacheinander durchlaufen.

Vor dem Compilieren durchsucht der Pre-Prozessor die Quelldatei vom Anfang bis zum Ende nach bestimmten Direktiven und wandelt diese in g.ltigen Clipper-Quellcode um, der vom Compiler verarbeitet werden kann. Das Ergebnis der Bearbeitung durch den Pre-Prozessor wird als Quelle zum Compilieren verwendet. Durch Angabe der Option /P wird der Pre-Prozessor angewiesen, das Ergebnis seiner Arbeit in eine Datei mit der Standard-erweiterung .PPO (Pre-Processor Output) zu schreiben. Auf diese Weise erhalten Sie den f.r die Compilierungs-Phase verwendeten Quellcode. Die Angabe der Option /P ist dann besonders sinnvoll, wenn Sie die Direktiven #command und #translate zur Definition von benutzer-definierten Befehlen (UDCs) verwendet haben. Aus der .PPO-Datei ersehen Sie, ob die .bersetzung der Befehle den Erwartungen entsprechend erfolgt ist.

In der folgenden Tabelle sind alle Pre-Prozessor-Direktiven zusammengefa.t. Ausf.hrliche Informationen zum Einsatz dieser Direktiven erhalten Sie im Kapitel Direktiven des Referenz-Handbuches I.

Die Arbeitsweise des Clipper-Compilers

Der Clipper-Compiler wandelt den in Clipper geschriebenen Quellcode in Intel-Objektcode-Dateien um. Die Quelldateien werden dabei in der Regel mit der DOS-Dateinamenserweitung .PRG gekennzeichnet, w.hrend Objektcode-Dateien immer die Dateinamenserweitung .OBJ erhalten. Diese Objekt-Dateien k.nnen dann mit einem Linker zu einem ausf.hrbaren Programm gebunden werden, das die Dateinamenserweitung .EXE erh.lt.

Alle Programm-, Format- und Prozedurdateien, auf die an irgendeiner Stelle der Programmdatei Bezug genommen wird, werden ebenfalls automatisch compiliert und mit in die resultierende Objekt-Datei aufgenommen. Diesen automatischen Vorgang k.nnen Sie durch Angabe der Option /M abstellen. Bei Script-Dateien hingegen erfolgt niemals eine automatische Compilierung solcher Dateien.

Eine Bezugnahme auf eine andere Datei kann aus einem der Befehle DO, SET FORMAT und SET PROCEDURE resultieren. Wenn der Aufruf von Clipper weder mit einer Script-Datei noch mit der Option /M erfolgt ist, so werden auch alle Dateien, die mit den o.g. Befehlen in der angegebenen Programmdatei aufgerufen werden, automatisch compiliert. Auch solche Dateien, die wiederum aus den automatisch compilierten Dateien aufgerufen werden, compiliert Clipper mit. Auf diese Weise werden schlie.lich alle miteinander verkn.pften Quelldateien verarbeitet. Dabei gelten folgende Regeln:

. Das in der Befehlszeile angegebene Programm wird zuerst compiliert.

. In der aktuellen Programmdatei aufgerufene Prozedur- und Format-Dateien sowie mit DO aufgerufene Prozeduren werden in der angegebenen Reihenfolge compiliert.

. Mit DO aufgerufene und noch nicht compilierte Prozeduren werden als .PRG-Datei im aktuellen Verzeichnis gesucht und, falls gefunden, compiliert. Falls sie nicht gefunden werden, erfolgt die Ausgabe einer Warnung.

. Dem Compiler noch nicht bekannte, mit SET FORMAT aufgerufene Formate werden als Datei mit der Dateinamenserweitung .FMT im aktuellen Verzeichnis gesucht und bei erfolgreicher Suche compiliert. Ist eine Prozedur mit dem gleichen Namen einer bei einem SET FORMAT-Befehl angegebenen Formatdatei bereits compiliert worden, so wird diese Prozedur bei Ausf.hrung des SET FORMAT-Befehles verwendet. In einem solchen Fall wird nicht nach einer Formatdatei gleichen Namens auf der Platte gesucht.

Bei Verwendung einer Script-Datei wird jede aufgef.hrte Programmdatei entsprechend den genannten Regeln compiliert, mit einer Ausnahme: Mit DO, SET FORMAT oder SET PROCEDURE aufgerufene Dateien werden nicht automatisch mit compiliert. Diese Dateien m.ssen explizit in die Dateiliste der Script-Datei aufgenommen werden.

Eine mit DO aufgerufene Prozedur mu. als .PRG-Datei desselben Namens im aktuellen DOS-Verzeichnis zug.nglich sein. Wird eine Prozedur als Funktion aufgerufen (in der Form TestProzedur()) und ist sie dem Compiler nicht bekannt, so wird nicht nach einer Datei gleichen Namens gesucht. Stattdessen vermutet Clipper, da. es sich um ein externes Symbol handelt. Der Compilierungs-Proze. wird in solch einem Fall ohne Ausgabe einer Warnung normal fortgesetzt.

Die Batch-Datei zum Compilieren und Linken

Bei normaler Installation des Clipper 5.0-Entwicklungssystems wird eine Batch-Datei CL.BAT als Beispiel zum Compilieren und Linken im Verzeichnis \CLIPPER5\BIN angelegt. Mit dieser Batch-Datei k.nnen Sie eine einzelne Programmdatei compilieren und, wenn kein Fehler aufgetreten ist, zu einer .EXE-Datei linken. Tritt ein Compiler-Fehler auf, so endet Clipper mit einem DOS-R.ckgabecode von 1 und es erfolgt kein Link-Proze..

Die Batch-Datei wird wie folgt aufgerufen:

```
CL <QuellDatei>
```

<QuellDatei> ist der Name der Programmdatei, die compiliert und gelinkt werden soll. Der Dateiname kann zus.tzlich einen Laufwerksbezeichner und einen Pfadnamen enthalten. Wird allerdings eine Dateinamenserweitung angegeben, so tritt ein Linker-Fehler auf!

Header-Dateien

Header-Dateien, oft auch als Include-Dateien bezeichnet, enthalten unter Clipper Pre-Prozessor-Direktiven und Befehlsdefinitionen. Header-Dateien besitzen keine vorgegebene Dateinamenserweitung und werden .ber die Pre-Prozessor-Direktive #include oder die Compiler-Option /U angegeben.

Die DOS-Umgebungsvariable INCLUDE kann zus.tzlich den Pfad angeben, in dem Header-Dateien zus.tzlich zum angemeldeten Verzeichnis gesucht werden sollen, z.B.:

SET INCLUDE=<PfadSpezifikation>

In dem mit INCLUDE angegebenen Pfad wird immer dann nach einer Header-Datei gesucht, wenn sie im aktuellen Verzeichnis nicht gefunden worden ist.

Bei der Installation von Clipper 5.0 mit den vorgegebenen Konfigurations-daten wird INCLUDE auf \CLIPPER5\INCLUDE eingestellt. Diese Einstellung k.nnen Sie .ndern, indem Sie die entsprechende Zeile in der Datei AUTOEXEC.BAT ab.ndern. Um z.B. die Suche nach Header-Dateien im Verzeichnis \INCLUDE auf dem Festplatten-Laufwerk D: durchf.hren zu lassen, m..te folgende Zeile in Ihrer AUTOEXEC.BAT-Datei vorhanden sein:

SET INCLUDE=D:\INCLUDE

Achtung
Bei der Suche nach Header-Dateien durchsucht Clipper zuerst das angemeldete Verzeichnis, danach das mit der Option /I angegebene Verzeichnis und schlie.lich den mit INCLUDE spezifizierten Pfad. In gewisser Weise wird der Pfadliste aus der Umgebungsvariablen INCLUDE mit der Option /I ein zus.tzlicher Suchpfad vorangestellt.

See Also:

Erzeugte Dateitypen

Der Clipper-Compiler kann drei verschiedene Typen von Ausgabedateien erzeugen. Der Name einer Ausgabedatei kann dabei einen kompletten oder aber teilweisen Pfad enthalten, so da. Clipper angewiesen wird, die Ausgabedatei explizit in diesen Pfad zu schreiben. Ohne Angabe eines Pfades wird die Ausgabedatei im aktuellen Verzeichnis abgelegt.

Objekt-Dateien

Das Ergebnis einer erfolgreichen Compilierung ist eine einzelne Objekt-Datei. Der vorgegebene Name der so erzeugten Datei entspricht dem Namen der angegebenen Script- oder Programmdatei, allerdings mit der neuen Erweiterung .OBJ. Sie k.nnen aber mit der Compiler-Option /O einen anderen Namen und ein anderes Verzeichnis spezifizieren. N.here Informationen hierzu finden Sie im Abschnitt Compiler-Optionen dieses Kapitels.

Tempor.re Dateien

W.hrend des Compilierungs-Prozesses legt Clipper in der Regel eine oder mehrere tempor.re Dateien an. Sie k.nnen mit der DOS-Umgebungsvariablen TMP bestimmen, in welchem Pfad diese tempor.ren Dateien angelegt werden, z.B.:

```
SET TMP=<PfadName>
```

Tempor.re Dateien werden nach Erf.llung ihres Zwecks von Clipper automatisch wieder gel.scht. Daher sind sie in der Regel f.r den Anwender nicht sichtbar. Die TMP-Variable dient nur dazu, ein anderes Laufwerk und einen anderen Pfad f.r die tempor.ren Dateien anzugeben, falls das aktuelle Laufwerk, das normalerweise zur Ablage der tempor.ren Dateien verwendet wird, nicht mehr gen.gend Platz bietet. Ein anderer Zweck der TMP-Variablen kann die Beschleunigung des Compilierungs-Prozesses sein, indem eine RAM-Disk als Ziellaufwerk f.r die tempor.ren Dateien spezifiziert wird. Durch das unmittelbare L.schen der TMP-Dateien wird in jedem Fall kein wertvoller Platz auf der Platte verschwendet.

Wenn Sie beispielsweise auf Laufwerk C: arbeiten und das Verzeichnis \TEMP auf Laufwerk D: als Verzeichnis f.r die tempor.ren Dateien verwenden m.chten, geben Sie ein:

```
SET TMP=D:\TEMP
```

Achtung

Mit der Compiler-Option /T kann ebenfalls ein Pfad f.r die tempor.ren Dateien angegeben werden. Falls sowohl /T als auch SET TMP gleichzeitig verwendet werden, hat die Compiler-Option /T Vorrang.

Ausgabe-Listing des Pre-Prozessors

Durch Angabe der Option /P k.nnen Sie ein Listing der Pre-Prozessor-Ausgabe in eine Datei schreiben. Der vorgegebene Name der so erzeugten Listing-Datei entspricht dem Namen der angegebenen Script- oder Programmdatei, jedoch mit der neuen Erweiterung .PPO.

See Also:

Vorbereitungen des Programmes f.r den Debugger

Um alle M.glichkeiten der Clipper Debuggers ausnutzen zu k.nnen, m.ssen Vorbereitungen getroffen werden. Dieser Abschnitt beschreibt eventuell n.tige .nderungen des Quelltextes sowie den Compiliervorgang.

. Compilation des Quelltextes

Der erste Schritt, um ein Programm mit dem Debugger zu bearbeiten, ist die Compilation des Quelltextes mit der /B-Option. Diese Option weist den Compiler an, die n.tigen Debugger-Informationen in die .OBJ-Datei einzuf.gen.

```
CLIPPER MeinProg /B
```

In diesem Fall werden alle, von MeinProg aufgerufenen Unterprogramme, ebenfalls mit aktiver /B-Option compiliert. Wenn Sie allerdings .CLP-Dateien benutzen, oder jedes Programm Ihrer Applikation seperat compilieren (/M), m.ssen Sie die /B-Option bei jedem Compileraufruf mit angeben. Sonst wird, w.hrend Sie mit dem Debugger arbeiten, jedes Unterprogramm, das ohne diese Option compiliert wurde ignoriert.

*** Achtung ***

Um den Clipper-Debugger benutzen zu k.nnen, m.ssen Ihre Applikationen mit Clipper 5.0 compiliert werden.

. Programmzeilen mit mehreren Anweisungen

Clipper 5.0 erlaubt es, mehrere Programmanweisungen in einer einzigen Zeile zu schreiben:

```
nSeiteNeu := (( ++nZeile % 55) == 0); Qout(nZeile)
```

Wenn man im Debugger mit Hilfe der Cursor-Tasten den Positionszeiger eine Zeile vor oder zur.ck bewegt, wird immer eine komplette Zeile .bersprungen, egal wieviel Anweisungen diese Zeile enth.lt. Im obigen Beispiel l..t der Debugger es nicht zu, da. Sie die beiden Anweisungen nacheinander ausf.hren oder einen Breakpoint dazwischen setzen. Dies gesamte Zeile wird als eine Einheit betrachtet.

Der Quelltext im obigen Beispiel sollten also wie folgt in zwei Zeilen zerlegt werden:

```
nSeiteNeu := (( ++nZeile % 55) == 0)
Qout(nZeile)
```

Dadurch wird das Debuggen einfacher und der Quelltext lesbarer.

. Header-Dateien

Header-Dateien werden im Programm mit Hilfe der #include-Direktive angesprochen. .blicherweise enthalten Sie sogenannte Symbolische Konstanten und Pseudofunktionen, sie k.nnen jedoch auch normalen Programmcode enthalten.

Um eine Header-Datei zu betrachten, k.nnen Sie .ber [ALT]+[F] das File-Men. ansprechen, um dann den Men.punkt View auszuw.hlen. In der dann erscheinenden Dialogbox geben Sie den Namen der gew.nschten Header-Datei ein und best.tigen den Namen mit [RETURN]. Der Quelltext der gennanten Header-Datei wird im Code-Window angezeigt.

. Symbolische Konstanten und Pseudofunktionen

Konstanten und Pseudofunktionen, die mit der #define-Direktive des Pre-Prozessors erzeugt wurden, k.nnen nicht auf dieselbe Art und Weise wie Ausdr.cke und Variablen beobachtet werden. Pre-Prozessor-Direktiven existieren nur f.r den Pre-Prozessor und nicht mehr zur Laufzeit. Wenn

Sie Ihr Programm mit der /P-Option compiliert haben, wird eine Datei mit .PPO-Erweiterung erzeugt. Diese Datei enth.lt die Ausgabe des Pre-Prozessors und kann mit dem Men.punkt Preprozessed Code im View-Men. angezeigt werden. Der aktuelle Quelltext bleibt im Code-Window sichtbar, w.hrend die Pre-Prozessor-Ausgabe jeweils unter der Quellcode-Zeile erscheint.

. **Code-Bl.cke**
Ein Code-Block enth.lt ausf.hrbaren Programmcode, der von Clipper 5.0 wie normale Daten behandelt wird und z.B. als Parameter an andere Programmteile .bergeben werden kann. Dieser Programmcode wird zur Compilierzeit .bersetzt, daher k.nnen Sie nur den Namen des Code-Blocks im Debugger betrachten, nicht den dahinterstehenden Quelltext. Allerdings k.nnen Sie Code-Bl.cke jederzeit innerhalb des Debuggers ausf.hren:

```
? EVAL(bBlock)
```

Sobald Sie diese Anweisungen im Command-Window eingeben, wird der Code-Block bBlock ausgef.hrt. Jeder Code innerhalb von bBlock wird ausgef.hrt und angezeigt.

Wenn Sie einen Quelltext debuggen, der Code-Bl.cke enth.lt, wird die Deklaration des Code-Blockes jedesmal angezeigt, sobald der Code-Block aufgerufen wird. Dadurch k.nnen Sie den Inhalt des Code-Blocks erkennen, unabh.ngig davon, ob der Block in der aktuelle Routine deklariert wurde oder nicht.

. **Makro-Ersetzung**
Der Debugger behandelt Makros wie normale Zeichenketten. So k.nnen diese mit inspiziert oder mit dem ?-Befehl betrachtet werden. So zeigt z.B.:

```
? &cMakro
```

den Inhalt des expandierten Makros cMakro an.

Wenn die Makro-Variable ienen Code-Blaock anspricht, wird der ?-Befehl nur das Code-Block Symbol {||..} anzeigen. Der tats.chliche Programmcode innerhalb des Code-Blocks wurde bereits zur Compilierzeit .bersetzt und ist f.r den Clipper-Debugger nicht verf.gbar.

Der Aufruf des Debuggers

Um den Debugger von der DOS-Kommandozeile her aufzurufen gilt folgende Syntax:

CLD [<Optionen>] <exeDatei>

CLD.EXE ist der ausf.hrbare Clipper-Debugger. Wenn Sie das Clipper 5.0 Entwicklungssystem mit der Standard-Konfiguration installiert haben, sollte sich CLD.EXE im Verzeichnis \CLIPPER5\BIN befinden und Ihre DOS-Pfadangabe korrekt gesetzt sein.

<exeDatei> ist der Name der ausf.hrbaren Datei (.EXE), die Sie debuggen m.chten. Wenn diese Datei nicht mit der /B-Option compiliert wurde, k.nnen Sie nicht mit dem gewohnten Komfort des Debugger arbeiten. Es k.nnen zwar watchpoints und Tracepoints definiert werden, allerdings wird kein Quelltext angezeigt. Haben Sie beim compilieren zus.tzlich die /L-Option angegeben, kann die Applikation nur ausgef.hrt werden, und Ihnen stehen keine Debugger Funktionen zur Verf.gung.

Um den Debugger innerhalb der Ausf.hrung einer Applikation kann er mit der Tastenkombination [ALT+[D] aktiviert werden.

Bei Aufruf des Debuggers von der DOS-Kommandozeile k.nnen zus.tzlich einige Optionen angegeben werden.

/Quiet
Wird diese Option angegeben, wird die Applikation direkt gestartet und Sie k.nnen den Debugger mit der Tatenkombination [AKT]+[D] aktivieren.

/Slit (nur f.r VGA-Bildschirme)
Wird diese Option angegeben, schaltet der Debugger in den 50-Zeilen Textmodus und teilt den Bildschirm in ein Applikationsteil (obere Bildschirmh.lfte) und einen Debuggerteil (untere Bildschirmh.lfte). Dieser Modus hat der Vorteil, das Sie den Ablauf Ihrer Applikation in der oberen Bildschirmh.lfte beobachten k.nnen und gleichzeitig die Debugging-Informationen im unteren Bildschirmteil verfolgen k.nnen.
Dieser Modus steht allerdings nur dann zur Verf.gung, wenn Sie mit einem VGA-Bildschirmadapter arbeiten.

/43 (nur f.r EGA/VGA-Bildschirme)
Diese Option schaltet den Bildschirm in den 43-Zeilen Textmodus, damit sie mehr Informationen gleichzeitig auf dem Bildschirm darstellen k.nnen.

/50 (nur f.r VGA-Bildschirme)
Diese Option schaltet den Bildschirm in den 50-Zeilen Textmodus, damit sie mehr Informationen gleichzeitig auf dem Bildschirm darstellen k.nnen.

@<ScriptFile>
Diese Option erlaubt die Angabe einer Script-Datei bei Aufruf des Debuggers. Dies hat die gleiche Auswirkung wie die Eingabe eines INPUT-Kommandos w.hrend der Debugger aktiv ist.

AT() Bestimmt die Position von Zeichen in einer Zeichenfolge

Syntax

AT(<cSuchfolge>, <cZeichenfolge>) -> nPosition

Parameter

<cSuchfolge> ist die zu suchende Zeichenfolge.

<cZeichenfolge> ist die Zeichenfolge, in der die Position von <cSuchfolge> bestimmt wird.

R.ckgabe

AT() liefert die Position des ersten Auftretens von <cSuchfolge> in <cZeichenfolge>. Wenn <cSuchfolge> in <cZeichenfolge> nicht enthalten ist, wird 0 zur.ckgegeben.

Anwendung

AT() ist eine Zeichenfunktion, anwendbar, um die Position des ersten Auftretens einer Zeichenfolge in einer Zeichenkette zu bestimmen. Wenn Sie lediglich das Vorkommen einer Zeichenkette in einer anderen testen wollen, sollten Sie den Operator "$" benutzen. Wollen Sie das erste Auftreten von rechts her .berpr.fen, benutzen Sie die Funktion RAT().

Beispiele

. Die folgenden Beispiele zeigen, wie AT() arbeitet:

```
? AT("a", "abcde")         // Resultat: 1
? AT("bcd", "abcde")       // Resultat: 2
? AT("a", "bcde")          // Resultat: 0
```

. Dieses Beispiel teilt eine Zeichenkette an der Stelle, an der das Komma auftritt:

```
cString := "Meier, Susanne"
? SUBSTR(cString, 1, AT(",", cString) - 1)  // Resultat: "Meier"
? SUBSTR(cString, AT(",", cString) + 1)     // Resultat: "Susanne"
```

Dateien

Bibliothek ist CLIPPER.LIB

See Also:

ANIMATE Ausf.hrungsmodus auf Animation umschalten

Syntax

ANIMATE

Beschreibung

Der Befehl ANIMATE l..t eine Applikation im Animationsmodus ablaufen. Das bedeutet, da. der Debugger eine Programmzeile ausf.hrt, den Execution Bar zur n.chsten Programmzeile weiterbewegt und diese ausf.hrt. Wenn die Exchange Screen-Option eingeschaltet ist, wird die Programmausgabe jeder ausgef.hrten Zeile angezeigt.
Dieser Vorgang h.lt solange an, bis das Programm einen Breakpoint oder einen Tracepoint erreicht. Um die Geschwindigkeit des Ablaufes zu beeinflussen, dient das SPEED-Kommando oder die Option Step Speed im Run-Men.. Der Programmablauf stoppt bei einem beliebigen Tastendruck.
Zus.tzliche Informationen .ber Ausf.hrungsmodi finden Sie in Kapitel 4 Debuggen eines Programmes.

See Also:

BP Setzen eines Breakpoints

Syntax

BP [AT] <nZeilenNr> [[IN] <cDateiName>] | <cFunktionsName>

Parameter

<nZeilenNr> ist die Zeilennummer, an der die Programmausf.hrung stoppen soll

<cDateiName> bezeichnet die Quelltextdatei, in welcher der Breakpoint gesetzt werden soll. Wenn keine Dateinamenserweiterung angegeben wird, benutzt der Debugger automatisch .PRG

<cFunktionsName> ist der Name der Funktion, bei deren Aufruf der Programmablauf unterbrochen werden soll.

Beschreibung

BP definiert eine bestimmte Programmzeile oder einen Funktionsaufruf als Breakpoint. Dadurch wird die Programmausf.hrung angehalten, sobald diese Programmzeile erreicht, bzw. die Funktion aufgerufen wird. Mit dem DELETE-Befehl wird ein gesetzter Breakpoint wieder gel.scht.
Weitere Informationen .ber Breakpoints finden Sie in Kapitel 4 Debuggen eines Programmes.

Beispiele

. Die Eingabe von:

```
# BP 12 [Return]
```

markiert die Zeile 12 des aktuellen Programmes im Code Window invers. Alle Programmzeilen, die invers dargestellt werden, sind Breakpoints.

. Dieses Beispiel setzt die Zeile 15 in der Datei FILE2.PRG als Breakpoint:

```
# BP AT 15 in FILE2.PRG [Return]
```

. Hier wird eine UDF als Breakpoint eingesetzt, so da. die Programmausf.hrung stoppt, sobald diese Funktion aufgerufen wird:

```
# BP TestUDF [Return]
```

See Also:

CALLSTACK Kontrolle des Callstack Windows

Syntax

CALLSTACK on | OFF

Parameter

ON reduziert die Breite das Code Windows und der Callstack wird im Callstack Window am rechten Bildschirmrand angezeigt.

OFF schlie.t das Callstack Window und stellt die urspr.ngliche Gr..e des Code Windows wieder her.

Beschreibung

CALLSTACK bestimmt, ob der Callstack Window angezeigt wird oder nicht. Dieses Fenster enth.lt eine Liste aller pending activations mit der aktiven Routine als oberstes Element. Mehr Informationen .ber den Callstack finden Sie in Kapitel 4, Debuggen eines Programmes.

Beispiele

. Um das Callstack Window zu .ffnen:

```
# CALLSTACK ON [Return]
```

. Um das Callstack Window wieder zu schlie.en:

```
# CALLSTACK OFF [Return]
```

DELETE L.schen von Debugger-Einstellungen

Syntax

DELETE [ALL [WP|TP |BP |PP]] [WP |TP |BP |PP <nNummer>]

Parameter

ALL l.scht alle Debugger-Einstellungen

WP l.scht alle Watchpoints

TP l.scht alle Tracepoints

BP l.scht alle Breakpoints

PP l.scht alle Passpoints

<nNummer> ist die Nummer eines bestimmten Watch-, Trace-, Break- oder Passpoints, der gel.scht werden soll.

Beschreibung

DELETE l.scht einzelne oder alle Watch-, Trace-, Break- oder Passpoints. Eine genaue Beschreibung dieser Elemente finden Sie in Kapitel 4, Debuggen eines Programmes.
Wenn eine einzelne Einstellung gel.scht werden soll, mu. der entsprechende numerische Bezeichner des zu l.schenden Elementes angegeben werden. Bei Watch- oder Tracepoints ist dies die am linken Rand des Watch Windows stehende Nummer des Elementes. F.r Break- oder Passpoints mu. das LIST-Kommando eingesetzt werden, um die Nummer des Elementes zu bestimmen.

Beispiele

. Um einen Breakpoint zu l.schen, mu. zuerst die Nummer des l.schenden Breakpoints mit

```
# LIST BP [Return]
```

ermittelt werden. Wenn der zu l.schende Breakpoint z.B. die Nummer 3 hat, kann dieser dann mit

```
# DELETE BP 3 [Return]
```

gel.scht werden.

. In diesem Beispiel werden alle Watchpoints gel.scht:

```
# DELETE ALL WP [Return]
```

. Im n.chsten Beispiel werden alle Debugger-Einstellungen gel.scht:

```
# DELETE ALL [Return]
```

See Also:

DOS Laden der DOS-Kommandoshell, ohne den Debugger zu verlassen

---

Syntax

DOS

Beschreibung

DOS l.dt eine tempor.re Kopie von COMMAND.COM. Dadurch k.nnen DOS-Kommandos eingegeben werden, ohne den Debugger verlassen zu m.ssen. Zur R.ckkehr in den Debugger dient dann das DOS-Kommando "Exit". Beachten Sie, da. gen.gend Speicher zur Verf.gung stehen mu., um COMMAND.COM und weitere Programme, aufrufen zu k.nnen, anderenfalls wird eine Fehlermeldung ausgegeben. Da diese Kopie nur zeitweise geladen wird, verfallen alle Umgebungseinstellungen, die Sie dort vornehmen, sobald Sie in den Debugger zur.ckkehren.

FIND Sucht eine Zeichenkette

Syntax

FIND <cSuchString>

Parameter

<cSuchString> ist die zu suchende Zeichenkette

Beschreibung

FIND durchsucht das im Code Window angezeigte Programm nach dem spezifizierten Zeichenstring. Die Suche beginnt immer in der ersten Programmzeile, unabh.ngig von der aktuellen Position des Position Indicators. Falls der Suchstring gefunden wird, bewegt der Debugger den Position Indicator in die entsprechende Programmzeile, anderfalls wird eine Fehlermeldung ausgegeben.
Dieser Befehl beachtet die Einstellung des Case Sensitiv Schalters im Locate-Men.. Wenn die Option eingeschaltet ist (gekennzeichnet durch ein H.kchen), beachtet FIND beim Suchvorgang auch die korrekte Gro.- und Kleinschreibung, ansonsten gen.gt eine Zeichenkette des gleichen Buchstabenmusters.

Beispiele

Wenn der vermutete Fehler darin begr.ndet sein k.nnte, da. die entsprechende Datenbank nicht ge.ffnet wurde, kann nach dem Schl.sselwort "USE" gesucht werden:

```
# FIND USE [Return]
```

See Also:

GO F.hrt die Applikation im Ablaufmodus aus

Syntax

GO

Beschreibung

GO schaltet auf den Ausgabebildschirm der Applikation um und startet diese. Die Applikation l.uft solange, bis ein Breakpoint oder Tracepoint erreicht wird, ohne da. zwischendurch in den Debugger-Bildschirm zur.ckgeschaltet wird. Um die Programmkontrolle w.hrend dieses Modus wieder an den Debugger zu .bergeben, und ins Code Window zur.ckzukehren, dient die Tastenkombination [ALT]+[D]. Ein nochmaliger Aufruf von GO startet das Programm von der aktuellen Programmzeile aus und f.hrt die Programmausf.hrung bis zur n.chsten Abbruchbedingung fort. Mehr Informationen .ber Ablaufmodi, Breakpoints und Tracepoints finden Sie in Kapitel 4, Debuggen eines Programmes.

Beispiele

. Angenommen, im Quelltext wurden Breakpoints in Zeile 15 und Zeile 30 gesetzt. Die Eingabe von:

```
# GO [Return]
```

f.hrt das Programm aus, bis Zeile 15 erreicht wird. Ein dann einzugebenes weiteres GO startet das Programm erneut bis Zeile 30.

GOTO Bewegt den Position Indicator zu einer bestimmten Programmzeile

---

Syntax

GOTO <nZeilenNummer>

Parameter

<nZeilenNummer> ist die Nummer der Programmzeile, zu der der Position Indicator bewegt werden soll.

Beschreibung

GOTO bewegt den Position Indicator von seiner aktuellen Position zu der angegebenen Zeilennummer. Wenn eine ung.ltige Zeilennummer eingegeben wird (z.B. eine Zahl die gr..er ist, als die komplette Anzahl von Programmzeilen), bewegt sich der Position Indicator nicht, und es wird eine Fehlermeldung ausgegeben.
GOTO bewegt den Position Indicator zur angegebenen Zeile, unabh.ngig davon, ob die Zeilennummern im Code Window angezeigt werden oder nicht. Allerdings ist es zu empfehlen, die Zeilennummern mit anzeigen zu lassen, da der Quelltext dann lesbarer wird.

Beispiele

Dieses Beispiel bewegt den Position Indicator zur Programmzeile 30 des aktuell angezeigten Programmes:

```
# GOTO 30 [Return]
```

See Also:

HELP Anzeige des Hilfefensters

Syntax

HELP

Beschreibung

HELP .ffnet das Help Window, in dem eine Liste der Funktionstastenbelegung und eine Kurzbeschreibung der verf.gbaren Kommandos angezeigt wird. Innerhalb der Kommandobeschreibung kann mit den Tasten [PfeilAuf], [PfeilAb] zwischen den einzelnen Befehlen ausgew.hlt werden. Durch [ESC] wird das Help Window verlassen und in den Debugger zur.ckgekehrt.
Weitere Informationen .ber das Help Window finden Sie in Kapitel 3, Der Debugger-Bildschirm.

INPUT Einlesen von Kommandos aus einer Script-Datei

Syntax

INPUT <cDateiName>

Parameter

<cDateiName> ist der Name einer Script-Datei, aus der die Kommandos ausgelesen werden sollen. Wird keine Dateinamenserweiterung angegeben, setzt der Debugger die Erweiterung .TXT ein.

Beschreibung

Nach INPUT liest der Debugger die Kommandos aus der angegebenen Script-Datei ein, und f.hrt diese aus. Eine vollst.ndige Beschreibung von Script Dateien finden Sie in Kapitel 2, Aufruf des Debuggers.

Beispiele

. Um die Kommandos in der Script Datei "ZeigBugs.txt" auszuf.hren, gen.gt die Eingabe von:

INPUT ZEIGEBUGS.TXT [Return]

INSPECT CALLSTACK Anzeige des Callstacks

Syntax

INSPECT CALLSTACK

Beschreibung

Durch INSPECT CALLSTACK wird das Callstack Window aktiviert und auf der rechten Seite des Bildschirmes angezeigt. Mit Hilfe der Cursortasten kann eine pending activation zur Ansicht ausgew.hlt. Der zur gew.hlten Routine geh.rige Quelltext wird dann im Code Window angezeigt, und der Position Indicator zeigt auf die zuletzt in dieser Routine ausgef.hrte Programmzeile. Um das urspr.ngliche Programm zur.ckzuholen, dient das RESUME-Kommando, bzw. die Resume-Option im File-Men..

Falls die ausgew.hlte Routine ein Code Block sein sollte (ein Code Block wird durch ein vorgestelltes "b" gekennzeichnet), wird im Callstack der Name der Routine angezeigt, in der dieser Block erzeugt wurde. Im Code Window erscheint der dazugeh.rige Quelltext.

Mit der Taste [TAB] wird das Callstack Window deaktiviert. Eine Tabelle der im Code Window g.ltigen Tastenkombination finden Sie in Kapitel 3, Der Debugger-Bildschirm.

INSPECT WP Ver.ndern und Anzeigen eines Watchpoints

Syntax

INSPECT WP

Beschreibung

INSPECT WP aktiviert das Watch Window. Hier k.nnen Werte von Feld- bzw. Speichervariablen oder Ausdr.cken untersucht und ge.ndert werden. Sobald dieses Kommando abgesetzt wurde, kann mit einem Auswahlbalken ein Objekt im Watch Window ausgew.hlt werden. Der aktuelle Wert dieser Feld- oder Speichervariablen bzw. des Ausdruckes wird dann in einer Dialogbox angezeigt und kann dort ver.ndert werden. Um die Dialogbox ohne .nderung des Wertes zu verlassen, kann dieser Vorgang mit [ESC] abgebrochen werden.
Wenn die zu .ndernde Variable noch keinen definierten Wert besessen hat, wird dieser erzeugt und der Variablen zugewiesen.
Um das Watch Window wieder zu deaktivieren, gen.gt das Bet.tigen von [TAB]. Dadurch wird das n.chste Fenster (entweder das Code- oder das Callstack Window) aktiviert. Genauere Informationen .ber die Navigation im Watch Window finden Sie in Kapitel 3, Der Debugger Bildschirm.

LIST Auflisten von Watch-, Trace-, Break- und Passpoints

Syntax

LIST WP|TP |BP |PP

Parameter

WP listet alle Watchpoints

TP listet alle Tracepoints

BP listet alle Breakpoints

PP listet alle Passpoints

Beschreibung

LIST zeigt alle Watch-, Trace-, Break- und Passpoints an.

Beispiele

. Im Quelltext AddData.prg seien in den Zeilen 10 und 15 Breakpoints gesetzt, im Quelltext ViewData.prg in den Zeilen 2 und 6. Dann ergibt die Eingabe:

```
# LIST BP [Return]

1) at line 10 in (ADDDATA.PRG)
2) at line 15 in (ADDDATA.PRG)
3) at line  2 in (VIEWDATA.PRG)
4) at line  6 in (VIEWDATA.PRG)
```

. In diesem Beispiel werden alle Passpoints angezeigt:

```
# LIST PP [Return]
```

See Also:

NEW Aufruf einer neuen .EXE-Datei

Syntax

NEW <cExeDatei>

Parameter

<cExeDatei> ist der Name des ausf.hrbaren Programmes

Beschreibung

NEW l.dt eine neue .EXE Datei in den Debugger. Falls das angegebene Programm gefunden wird, ersetzt dieses die aktuelle Applikation und die Debugger-Einstellungenen werden gel.scht. Andernfalls gibt der Debugger eine Fehlermeldung aus.

NEXT Sucht das n.chste Auftreten eines Suchbegriffes

Syntax

NEXT

Beschreibung

NEXT sucht das n.chste Auftreten eines vorher mit FIND gefundenen Suchbegriffes. Falls kein weiterer Begriff gefunden wird oder der Position Indicator auf der letzten Programmzeile steht, gibt der Debugger eine Fehlermeldung aus.
Wie FIND beachtet auch NEXT die Einstellung von Case Sensitiv im Locate-Men..

See Also:

NUM Schaltet die Anzeige von Zeilennummern ein oder aus

Syntax

NUM ON | off

Parameter

ON schaltet die Anzeige der Zeilennummern im Code Window ein

OFF schaltete die Anzeige der Zeilennummern im Code Window aus

Beschreibung

NUM schaltet die Anzeige der Zeilennummern f.r den aktuellen Quelltext im Code Window um. Die Anzeige der Zeilennummern ist Standardeinstellung, sobald der Debugger aufgerufen wird. Diese Einstellung ist vor allem im Zusammenhang mit dem GOTO-Befehl sinnvoll, um den Position Indicator zu einer bestimmten Zeile zu bewegen.

OUTPUT Anzeige der Programmausgabe

Syntax

OUTPUT

Beschreibung

OUTPUT l.scht vor.bergehend den Debugger-Bildschirm und zeigt die Ausgabe des bearbeiteten Programmes an. Dadurch l..t sich die Programmausgabe auf Fehler untersuchen. Um in den Debugger zur.ckzukehren, kann eine beliebige Taste gedr.ckt werden.

PP Setzen eines Passpoints

Syntax

PP [AT] <nZeilenNr> [[IN] <cDateiName>]

Parameter

<nZeilenNr> ist die Zeilennummer, an der ein pass count angezeigt werden soll

<cDateiName> bezeichnet die Quelltextdatei, in der ein Passpoint gesetzt werden soll. Wenn keine Dateinamenserweiterung angegeben wird, benutzt der Debugger automatisch .PRG

Beschreibung

PP erzeugt einen Passpoint f.r eine bestimmte Programmzeile. Dadurch wird am Ende der Programmzeile der pass count angezeigt, es wird also gezeigt, wie oft diese Programmstelle bereits abgearbeitet wurde. Beachten Sie, da. der pass count erst nach der Abarbeitung der Programmzeile erh.ht wird. Um einen Passpoint zu l.schen, dient das DELETE-Kommando.
Mehr Informationen zu Passpoints finden Sie in Kapitel 4 Debuggen eines Programmes.

Beispiele

. Die Eingabe von:

```
# PP 267 [Return]
```

erzeugt einen Passpoint in Zeile 267 der aktuellen Programmdatei im Code Window.

. In diesem Beispiel wird in Zeile 30 der Datei DRUCK.PRG ein Passpoint gesetzt:

```
# PP AT 30 IN DRUCK.PRG [Return]
```

See Also:

PREV Sucht das vorherige Auftreten eines Suchbegriffes

Syntax

PREV

Beschreibung

PREV sucht das vorherige Auftreten eines vorher mit FIND gefundenen Suchbegriffes. Falls kein weiterer Begriff gefunden wird, oder der Position Indicator auf der ersten Programmzeile steht, gibt der Debugger eine Fehlermeldung aus.
Wie FIND beachtet auch PREV die Einstellung von Case Sensitiv im Locate-Men..

See Also:

QUIT Verlassen des Debuggers

Syntax

QUIT

Beschreibung

QUIT beendet den Debugger, schlie.t alle offenen Dateien und kehrt nach DOS zur.ck.

RESTART L.dt das aktuelle Programm erneut in den Debugger

Syntax

RESTART

Beschreibung

Mit RESTART wird die aktuelle Applikation neu in den Debugger geladen und kann erneut ausgef.hrt werden. Alle Debugger-Einstellungen bleiben bestehen. Dieser Befehl ist die einzige M.glichkeit, ein Programm, das bereits ausgef.hrt wurde, neu zu starten.

Beispiele

. In diesem Beispiel wird die aktuelle Applikation gestartet:

```
# GO [Return]
```

. Nachdem Fehler aufgetreten sind, mu. das Programm eventuell neu gestartet werden, um den Fehler genauer zu lokalisieren:

```
# RESTART [Return]
```

RESUME R.ckkehr in den vorherigen Quelltext

Syntax

RESUME

Beschreibung

RESUME l.scht die mit Hilfe des VIEW-Befehls betrachtete Programmdatei aus dem Code Window und l.dt die vorherige Datei. Alle eingestellten Break- oder Passpoints werden beibehalten.

Beispiele

. Angenommen, das aktuelle Programm im Debugger hei.t PRINTDAT.PRG und die folgende Zeile l.dt eine Datei REPORT.PRG ein:

```
# VIEW REPORT.PRG [Return]
```

. Dann kann folgenderma.en nach PRINTDAT.PRG zur.ckgekehrt werden:

```
# RESUME [Return]
```

See Also:

SPEED Kontrolliert die Ablaufgeschwindigkeit

Syntax

SPEED <nGeschwindigkeit>

Parameter

<nGeschwindigkeit> ist eine Zahl, die die Ablaufgeschwindkeit einstellt. Diese Zahl kann zwischen 0 und 9 liegen, wobei 0 die gr..t m.gliche, 9 die langsamste Geschwindigkeit, darstellt.

Beschreibung

SPEED kontrolliert die Ablaufgeschwindigkeit im Animationsmodus. Mehr Informationen .ber die Ablaufmodi finden Sie in Kapitel 4, Debuggen eines Programmes. Der Aufruf von SPEED ohne Parameter zeigt die aktuelle Einstellung an. Diese Einstellung ist auch im Run-Men. rechts von der Option Step Speed sichtbar.

See Also:

STEP Ausf.hrung einer Anzahl von Programmschritten

Syntax

STEP <nAnzahlProgrammZeilen>

Parameter

<nAnzahlProgrammZeilen> ist die Anzahl der Programmzeilen, die ausgef.hrt werden sollen.

Beschreibung

STEP f.hrt eine gegebene Anzahl von Programmzeilen aus und h.lt dann das Programm wieder an. .hnlich wie GO arbeitet auch STEP im Ablaufmodus und zeigt nur den Ausgabebildschirm an. Der Programmablauf wird erst unterbrochen, wenn die gegebene Anzahl von Zeilen abgearbeitet wurde, oder ein Breakpoint bzw. Tracepoint erreicht wurde.

Beispiele

. Dieses Beispiel f.hrt die n.chsten 20 Programmzeilen aus:

```
# STEP 20 [Return]
```

See Also:

TP Definiert eine Variable oder einen Ausdruck als Tracepoint

---

Syntax

TP <exp>

Parameter

<exp> ist eine Feld- bzw. Speichervariable, oder ein Ausdruck. Er kann von jedem Datentyp sein.

Beschreibung

TP definiert das eingegebene Objekt als Tracepoint. Name, Typ und Inhalt des Tracepoints werden am Listenanfang im Watch Window hinzugef.gt. Einen Tracepoint kann man sich als bedingten Breakpoint vorstellen. Sobald sich der Wert des Tracepoints .ndert wird die Programmausf.hrung angehalten.
Mehr Informationen .ber Tracepoints finden Sie in Kapitel 4, Debuggen eines Programmes.

Beispiele

. In diesem Beispiel stoppt die Programmausf.hrung, sobald sich der Wert von nSeite .ndert:

```
# TP nSeite [Return]
```

. Hier h.lt das Programm an, sobald die Dateiendebedingung erf.llt wird:

```
# TP EOF() [Return]
```

. Dieses Beispiel stoppt das Programm, sobald die Variable i gr..er 10 ist:

```
# TP i > 10 [Return]
```

See Also:

VIEW Ansicht einer beliebigen Textdatei

Syntax

VIEW <cDateiName>

Parameter

<cDateiName> ist der Name der Datei, die angezeigt werden soll.

Beschreibung

Mit VIEW k.nnen beliebige Dateien in den Debugger geladen werden. Das Kommando kann z.B. zur Einsicht von Header Dateien, die .ber die #include-Direktive in das Programm eingebunden wurden, benutzt werden. In der geladenen Datei k.nnen Breakpoints oder Passpoints gesetzt werden.

Beispiele

. Die Datei LISTDBF.PRG laden, obwohl die Datei DATEN.PRG im Code Window ist:

```
# VIEW LISTDBF.PRG [Return]
```

. Man kann auch weitere Dateien mit VIEW einladen:

```
# VIEW DRUCK.PRG [Return]
```

und sp.ter mit RESUME wieder in das Orginalprogramm DATEN.PRG zur.ckkehren.

See Also:

BIN2I() Wandelt eine 16-Bit Zahl mit Vorzeichen in einen numerischen Wert um

---

Syntax

BIN2I(<cSignedInt>) -> nWert

Parameter

<cSignedInt> ist eine 16-Bit Zahl mit Vorzeichen in Form einer Zeichenfolge von 2 Zeichen, wobei das niederwertige Byte dem ersten Zeichen entspricht. Werden mehr als zwei Zeichen angegeben, wird der Rest ignoriert.

R.ckgabe

Numerischer Wert im Integer-Format.

Anwendung

BIN2I() wird in Verbindung mit FREAD() benutzt, um eine 2-Byte Zeichenfolge, formatiert als 16-Bit Integer-Zahl mit Vorzeichen, in einen numerischen Wert zu konvertieren. Dies ist n.tzlich, wenn Sie fremde Datentypen lesen und numerische Datentypen verarbeiten wollen, die in ihrer urspr.nglichen Form enthalten sind.

Beispiele

Dieses Beispiel .ffnet eine Datenbank unter Verwendung von systemnahen Dateifunktionen und liest das Datum des letzten Updates ein. Das Ergebnis ist dasselbe wie das von LUPDATE().

```
#include "FileIO.ch"
//
DateiNr := FOPEN("Verkauf.dbf",FO_READ)
//
// Setzen des Zeigers auf Byte 1
FSEEK(DateiNr, 1, FS_SET)
//
//Lesen des Datums des letzten Update's
Jahr    := BIN2I(FREADSTR(DateiNr, 1)) + CHR(0)
Monat   := BIN2I(FREADSTR(DateiNr, 1)) + CHR(0)
Tag     := BIN2I(FREADSTR(DateiNr, 1)) + CHR(0)
//
? LTRIM(STR(Tag)), LTRIM(STR(Monat)), LTRIM(STR(Jahr))
FCLOSE(DateiNr)
```

Dateien

Bibliothek ist EXTEND.LIB, Quelldatei ist EXAMPLEA.ASM

See Also:

WP Definiert eine Variable oder einen Ausdruck als Watchpoint

---

Syntax

WP <exp>

Parameter

<exp> ist eine Feld- bzw. Speichervariable, oder ein Ausdruck. Er kann von jedem Datentyp sein.

Beschreibung

WP definiert das eingegebene Objekt als Watchpoint. Name, Typ und Inhalt des Tracepoints werden am Listenanfang im Watch Window hinzugef.gt. Ein Watchpoint ist das gleiche wie ein Tracepoint, nur das die Programmausf.hrung nicht unterbrochen wird, sobald sich der Wert des Watchpoints .ndert.
Mehr Informationen .ber Watchpoints finden Sie in Kapitel 4 Debuggen eines Programmes.

Beispiele

. Dieses Beispiel definiert die Variable KundenNr als Watchpoint:

```
# WP KundenNr [Return]
```

. Hier wird im Watch Window nach jeder abgearbeiteten Programmzeile die Datensatznummer des aktuellen Arbeitsbereiches angezeigt:

```
# TP RECNO() [Return]
```

Die Befehle von .RTLink

Die .RTLink-Dateien

Der Linker .RTLink ben.tigt zum korrekten Ablauf verschiedene Dateien, die nachfolgend mit einer kurzen Erl.uterung aufgef.hrt sind:

RTLINK.EXE

Dieses Programm wird zur Erstellung von ausf.hrbaren .EXE-Dateien oder .PLL-Library-Dateien verwendet. Die Suche nach RTLINK.EXE erfolgt gem.. den normalen DOS-Suchregeln.

RTLINKST.COM

Diese Datei enth.lt den Start-Up-Code, der f.r jede mit .RTLink erzeugte .PLL-Datei benutzt wird. Dieser Code wird in die .PLL-Datei eingebunden, so da. er bei jeder Ausf.hrung des darauf zugreifenden .EXE-Programms verf.gbar ist. Daher wird RTLINKST.COM nur bei Erzeugung einer .PLL-Datei mit der Option PRELINK ben.tigt, nicht aber beim Aufruf einer zugeh.rigen .EXE-Datei. RTLINKST.COM mu. sich im gleichen Verzeichnis wie RTLINK.EXE befinden.

RTLUTILS.LIB

Diese Library-Datei enth.lt den Manager f.r statische Overlays. RTLUTILS.LIB wird immer dann ben.tigt, wenn Sie mit .RTLink Programme erstellen, die eine der Linker-Optionen f.r statische Overlays verwenden.

RTLINK.DAT

Diese Datei enth.lt alle Fehlermeldungen von .RTLink. Au.erdem beinhaltet sie alle notwendigen Daten, um die FREEFORMAT-Eingaben zu erkennen und zu interpretieren. RTLINK.DAT mu. sich im gleichen Verzeichnis wie RTLINK.EXE befinden und bei jedem Aufruf von RTLINK.EXE verf.gbar sein.

RTLINK.HLP

Diese Datei enth.lt den Hilfetext, der beim Aufruf von .RTLink mit der Option /HELP auf dem Bildschirm ausgegeben wird. RTLINK.HLP mu. sich im gleichen Verzeichnis wie RTLINK.EXE befinden.
Bei Verwendung der vorgegebenen Installation des Clipper 5.0-Entwicklungs-systems befindet sich RTLUTILS.LIB im Verzeichnis \CLIPPER5\LIB, alle anderen .RTLink-Dateien werden im Verzeichnis \CLIPPER5\BIN abgelegt.

Der Aufruf von .RTLink

.RTLink kann zwei verschiedene Eingabeformen verarbeiten: POSITIONAL (kompatibel zum Eingabeformat des Microsoft-Linkers LINK.EXE) und FREEFORMAT (kompatibel zum Eingabeformat von Plink86-Plus). Diese beiden Formen unterscheiden sich in der Art und Weise, wie dem Linker die zu verarbeitenden Module und Optionen angegeben werden.

Die FREEFORMAT-Schnittstelle

Falls nicht anders konfiguriert, arbeitet .RTLink standardm..ig mit dem Eingabeformat FREEFORMAT. Zur Verwendung von .RTLink mit dem POSITIONAL-Format mu. die Konfiguration, wie im n.chsten Abschnitt beschrieben, ge.ndert werden. Die Syntax zum Aufruf von .RTLink .ber die FREEFORMAT-Schnittstelle ist wie folgt:

```
RTLINK [FILE <ObjektDateienListe>
  [OUTPUT <AusgabeDatei>]
  [LIBRARY <LibDateienListe>]
  [<LinkerOptionenListe>]]
```

oder bei Verwendung einer Scriptdatei:

```
RTLINK @<Scriptdatei> <LinkerOptionenListe>
```

Die POSITIONAL-Schnittstelle

Die Alternative zur FREEFORMAT-Schnittstelle ist die POSITIONAL-Schnittstelle. Wenn Sie das POSITIONAL-Format als Voreinstellung benutzen m.chten, definieren Sie die Umgebungsvariable RTLINKCMD wie folgt:

```
SET RTLINKCMD=/POSITIONAL
```

Sie k.nnen mit RTLINKCMD nat.rlich auch weitere Optionen angeben. Zus.tzliche Informationen zur Konfiguration von .RTLink finden Sie im Abschnitt Die Konfiguration und Voreinstellung von .RTLink. Wenn das POSITIONAL-Format f.r .RTLink gew.hlt worden ist, mu. die folgende Syntax verwendet werden:

```
RTLINK [<ObjektDateienListe>,
  [<AusgabeDatei>],
  [<MapDatei>],
  [<LibDateienListe>]
  [<LinkerOptionenListe>]][;]
```

oder bei Verwendung einer Scriptdatei:

```
RTLINK @<Scriptdatei> <LinkerOptionenListe>
```

BATCH | NOBATCH Ein- bzw. Ausschalten der Eingabeaufforderung

---

Die Voreinstellung dieser beiden, einander ausschlie.enden Linker-Optionen ist NOBATCH. Wenn Sie keine dieser Optionen angeben, wird NOBATCH angenommen. In diesem Modus fragt .RTLink nach einem Pfadnamen, wenn eine LIB-, OBJ- oder PLT-Datei nicht gefunden werden kann. Da NOBATCH die Vorgabe ist, brauchen Sie diese Option nur dann angeben, wenn Sie eine vorherige BATCH-Einstellung (z.B. in RTLINKCMD) .berschreiben wollen. Die Abk.rzung f.r NOBATCH ist NOBAT.

Die Angabe von BATCH h.lt den Linker davon ab, nach einem Pfad zu fragen, wenn eine zu verarbeitende Datei nicht gefunden worden ist. Dieses Verhalten wird als Batchmodus bezeichnet, da der Linker seine Arbeit nicht f.r weitere Eingaben unterbricht. Wurde eine Datei nicht gefunden, so bricht der Linker mit einer Fehlermeldung ab und der DOS-R.ckgabecode wird auf 1 gesetzt. Die Abk.rzung f.r BATCH ist BAT.

IGNORECASE | NOIGNORCASE Ein- bzw. Ausschaltung der case sensitive

---

Die Voreinstellung dieser beiden, einander ausschlie.enden Linker-Optionen ist IGNORECASE. Wenn Sie keine dieser Optionen angeben, wird IGNORECASE angenommen. In diesem Modus behandelt .RTLink Klein- und Gro.schreibung gleich. Zum Beispiel k.nnte ein Funktionsname MeineFunc auch als MEINfunc angegeben werden. Der Linker w.rde sich trotzdem auf ein und dasselbe Symbol beziehen. Da IGNORECAS die Vorgabe ist, brauchen Sie diese Option nur dann angeben, wenn Sie eine vorherige NOIGNORCASE-Einstellung (z.B. in RTLINKCMD) .berschreiben wollen. Die Abk.rzung f.r IGNORECASE ist IG.

NOIGNORECASE bewirkt, da. die Gro.- und Kleinschreibung f.r Symbol und Segmentnamen relevant ist (=case sensitiv). MeineFunc und MEINEfunc werden als zwei verschiedene Symbole interpretiert werden. Normalerweise wird diese Option f.r CLipper-Programme nicht verwendet, da Clipper die Bezeichnung nicht nach Gro.- und Kleinschreibung unterscheidet. Es kann aber Probleme beim Linken von Clipper-Programmen mit C- oder Assemblermodulen geben, die case sensitivity voraussetzen und daher die Verwendung von NOIGNORECASE bedingen. Die Abk.rzung f.r NOIGNORECASE IST NOIG.

FREEFORMAT | POSITIONAL Einstellung des .RTLink-Schnittstellenmodus

---

.RTLink kann zwei verschiedene Eingabeformen verarbeiten: POSITIONAL (kompatibel zum Eingabeformat des Microsoft-Linkers LINK.EXE) und FREEFORMAT (kompatibel zum Eingabeformat von Plink86-Plus). Diese beiden Formen unterscheiden sich in der Art und Weise, wie dem Linker die zu verarbeitenden Module und Optionen angegeben werden.

Die FREEFORMAT-Schnittstelle

Falls nicht anders konfiguriert, arbeitet .RTLink standardm..ig mit dem Eingabeformat FREEFORMAT. Zur Verwendung von .RTLink mit dem POSITIONAL-Format mu. die Konfiguration, wie im n.chsten Abschnitt beschrieben, ge.ndert werden. Die Syntax zum Aufruf von .RTLink .ber die FREEFORMAT-Schnittstelle ist wie folgt:

```
RTLINK [FILE <ObjektDateienListe>
  [OUTPUT <AusgabeDatei>]
  [LIBRARY <LibDateienListe>]
  [<LinkerOptionenListe>]]
```

oder bei Verwendung einer Scriptdatei:

```
RTLINK @<Scriptdatei> <LinkerOptionenListe>
```

Die POSITIONAL-Schnittstelle

Die Alternative zur FREEFORMAT-Schnittstelle ist die POSITIONAL-Schnittstelle. Wenn Sie das POSITIONAL-Format als Voreinstellung benutzen m.chten, definieren Sie die Umgebungsvariable RTLINKCMD wie folgt:

```
SET RTLINKCMD=/POSITIONAL
```

Sie k.nnen mit RTLINKCMD nat.rlich auch weitere Optionen angeben. Zus.tzliche Informationen zur Konfiguration von .RTLink finden Sie im Abschnitt Die Konfiguration und Voreinstellung von .RTLink. Wenn das POSITIONAL-Format f.r .RTLink gew.hlt worden ist, mu. die folgende Syntax verwendet werden:

```
RTLINK [<ObjektDateienListe>,
  [<AusgabeDatei>],
  [<MapDatei>],
  [<LibDateienListe>]
  [<LinkerOptionenListe>]][;]
```

oder bei Verwendung einer Scriptdatei:

```
RTLINK @<Scriptdatei> <LinkerOptionenListe>
```

DYNAMIC **Ablage von Modulen in dynamische Overlays**

Die Option DYNAMIC ist das Gegenst.ck zu RESIDENT und bildet die Voreinstellung. Wenn Sie keine dieser Option angeben, wird DYNAMIC angenommen. Die Abk.rzung von DYNAMIC ist DYN.

Syntax

/DYNAMIC [:<OverlayDatei>]
DYNAMIC [INTO <OverlayDatei>]

Durch die Angabe von DYNAMIC wird .RTLink angewiesen, alle innerhalb der <ObjektDateienListe> oder <LibDateienListe> aufgef.hrten Clipper-Module als dynamische Overlays anstatt als residente Module zu linken. Diese M.glichkeit ist nur f.r Dateien vorhanden, die mit Clipper 5.0 compiliert wurden. In anderen Sprachen (z.B. C oder Assembler) erstellte Module werden immer als residente Module oder statische Overlays gelinkt.

Wenn f.r <OverlayDatei> keine Angaben gemacht wird, werden alle Module in die .EXE-Datei aufgenommen. Ist ein Name f.r <OverlayDatei> angegeben, dann werden alle geladenen Clipper-Module in diese <OverlayDatei> geschrieben. Die vorgegebene Extension f.r die <OverlayDatei> ist .OVL. Wenn die Option DYNAMIC mehrfach mit demselben Namen f.r <OverlayDatei> angegeben wird, werden die entsprechenden Clipper-Module nacheinander an die spezifizierte Overlay-Datei angehangen.

Da DYNAMIC die Vorgabe ist, mu. diese Option nur dann angegeben werden, wenn vorherige RESIDENT-Einstellung (Z.B. RTLINKCMD) .berschrieben werden soll. Dynamische Overlays werden bei Verwendung der Voreinstellung intern in die .EXE-Datei abgelegt. Sollen die dynamischen Overlays in eine externe Datei geschrieben werden, so mu. explizit DYNAMIC <OverlayDatei> angegeben werden.

RESIDENT Die Ablage von Modulen im Root-Bereich

RESIDENT ist das Gegenteil von der Linker-Option DYNAMIC. RESIDENT kann mit RES abgek.rzt werden. Durch RESIDENT werden alle nachfolgend angegebene Clipper-Module im Root-Bereich (Programmstamm) des aktuellen Programmes oder der augenblicklich erzeugten Pre-Link-Library abgelegt. Bei Angabe dieser Option wird die automatische Erzeugung von dynamischen Overlays aus Clipper-Modulen aufgehoben.

Beispiele

Sie k.nnen in einer .RTLink-Befehlszeile verschiedene Module als DYNAMIC oder RESIDENT linken, wenn Sie diese Optionen an der gew.nschten Stelle in der Liste der Objektdateien einf.gen.

Ein Beispiel f.r die POSITIONAL-Syntax:

```
RTLINK /DYNAMIC Test1 /RESIDENT Test2 /DYNAMIC:NeueOvl1 Test3;
```

Bei der Verwendung der FREEFORMAT-Syntax kann dasselbe mit einer Script-datei erreicht werden:

```
TRLINK @Script
```

Die Scriptdatei SCRIPT.LNK h.tte dabei folgenden Inhalt:

```
DYNAMIC
FILE Test1
RESIDENT
FILE Test2
DYNAMIC INTO NeueOvl1
FILE Test3
```

DEBUG **Zeigt das Laden von statischen Overlays zur Laufzeit an**

DEBUG stellt eine Methode dar, um Programme zu kontrollieren, die statische Overlayd verwenden. Wurde diese Option beim Linken angegeben, so zeigt der Overlay-Manager w.hrend der Applikations-Ausf.hrung den Namen jedes in den Speicher nachgeladenen Overlays an. Die Abk.rzung f.r DEBUG ist DEB.

HELP **Auflistung der .RTLink-Optionen**

HELP zeigt alle f.r .RTLink verf.gbaren Optionen auf dem Bildschirm an. Nach Ausgabe der m.glichen Optionen der m.glichen Optionen endet .RTLink mit dem R.ckgabecode 0 und erzeugt keine Ausgabedatei. Die Abk.rzung f.r HELP ist H.

MAP **Auflistung der PUBLIC-Symbole**

MAP bewirkt, da. .RTLink eine Map-Datei erzeugt, die eine oder mehrere Auswertungen und Berichte .ber den aktuellen Link-Prozess enth.lt. Die Ab-k.rzung f.r MAP ist MA.

Syntax

```
/MAP [:<MapOptionsListe>]
MAP [= <MapDatei>] [<MapOptionsListe>]
```

<MapDate> ist der Name der zu erzeugenden Datei. Im POSITIONAL-Format wird der Name der Map-Datei in der Befehlszeile angegeben und ist daher nicht in der Syntax enthalten. Falls kein Name aufgef.hrt worden ist, er-h.lt die Map-Datei denselben Namen wie die Ausgabedatei, jedoch mit der vorgegebenen Extension .MAP.

<MapOptionsListe> ist eine Liste von MAP-Optionen, die angeben, welche Auswertungen in die Map-Datei geschrieben werden sollen. Falls die Liste nicht aufgef.hrt ist, werden alle Berichte in die Map-Datei aufgenommen. Ansonsten kann eine Kombination der in der folgenden Tabelle aufgelisteten Buchstaben f.r die verschiedenen Berichtsarten angegeben werden:

Map-Report-Optionen

Option	Bedeutung
S	Listet die Segmente mit den zugeordneten Adressen auf
N	Listet die Public-Symbole mit ihren Adressen auf, Sortierfolge sind die Symbolnamen
A	Listet die Public-Symbole mit ihren Adressen auf, Sortierfolge sind die Adressen

SILENT Eingabeaufforderung f.r Script-Modus unterdr.cken

SILENT unterdr.ckt die Ausgabe der Linker-Eingabeanforerungen und -Antworten, wenn sich .RTLink im Scriptdatei-Modus befindet. Die Abk.rzung d.r SILENT ist SI.

VERBOSE **Ausgabe von Informationen des Link-Prozesses**

VERBOSE bewirkt, da. .RTLink w.hrend des Link-Prozesses kurze Statusmeldungen ausgibt, um anzuzeigen, welche Module gerade in welcher Reihenfolge gelinkt werden. W.hrend des incrementalen Linkens zeigt .RTLink bei Angabe von VERBOSE den aktuellen Prozentsatz, des in der .EXE-Datei verwendeten Speicherplatzes an. Die Abk.rzung f.r VERBOSE ist VE.

Syntax

VERBOSE [:<Ebene>]

<Ebene> kann 0,1 oder 2 sein. Mit diesem Parameter wird die Menge (Ausf.hrlichkeit) der ausgegebenen Informationen festgelegt.

INCREMETAL Ein- und Ausschalten von inkrementalem Linken
NOINCRELEMTAL

Die Voreinstellung dieser beiden, einander auschlie.enden Linker-Optionen ist NOINCREMENTAL. Wenn Sie keine dieser Optionen angeben, wird NOINCREMETAL angenommen. Da NOINCREMENTAL die Vorgabe ist, brauchen Sie diese Option nur dann angeben, wenn Sie eine vorherige INCREMENTAL-Einstellung (z.B. in RTLINKCMD) .berschrieben werden soll.

NOINCREMENTAL deaktiviert das inkrementale Linken von Clipper-Modulen, so da. das gesamte Programm neu gelinkt wird, unabh.ngig davon, ob sich Module ge.ndert haben oder nicht. Die Abk.rzung f.r NOINCREMENTAL ist NOIN.

INCREMENTAL kann mit INC abgek.rzt werden. Diese Option aktiviert das inkremenale Linken von Clipper-Modulen. Einfacher gesagt, .berpr.ft .RTLink beim inkrementalen Linken, welche Module seit dem letzten Link-Prozess ge.ndert worden sind und linkt dann nur diese Module neu. Dadurch geht ein Link-Prozess wesentlich schneller vonstatten.

DEFAULTLIBRARYSEARCH Ein- bzw. Ausschalten der vorgegebenen Librarysuche
NODEFAULTLIBRARYSEARCH

Die Voreinstellung dieser beiden, einander auschlie.enden Linker-Optionen ist DEFAULTLIBRARYSEARCH. Wenn Sie keine dieser Optionen angeben, wird DEFAULTLIBRARYSEARCH angenommen. Da DEFAULTLIBRARYSEARCH die Vorgabe ist, brauchen Sie diese Option nur dann angeben, wenn Sie eine vorherige NODEFAULTLIBRARYSEARCH-Einstellung (z.B. in RTLINKCMD) .berschrieben werden soll. Die Abk.rzungen f.r diese beiden Optionen sind DEFA bzw. NOD.

Compiler schlie.en oft die Namen der vorgegebenen Library-Dateien mit in die Objektdateien mit ein, so da. diese Library-Dateien vom Linker automatisch gesucht werden. Clipper 5.0 schlie.t z.B. die Dateien CLIPPER.LIB, EXTEND.LIB und DBFNTX.LIB ein. Diese Dateien m.ssen beim Linken nicht mehr als zu verarbeitende Library-Dateien angegeben werden. DEFAULTLIBRARYSEARCH veranla.t .RTLink, die Verweise auf diese Dateinamen innerhalb der Objektdateien zu benutzen.

NODEFAULTLIBRARYSEARCH veranla.t den Linker, die in den Objektdateien eingeschlossenen Dateinamen zu ignorieren. Falls nach dem Laden aller Module aus der <ObjektDateienListe> noch nicht definierte (undefined) Symbole gibt und keine weiteren Library-Dateien als Suchdateien angegeben sind, gibt .RTLink eine Liste aller nicht definierten Symbole aus.

EXTDICTIONARY Ein- bzw. Auschalten der Suche mit dem Extended Dictionary
NOEXTDICTIONARY

Die Voreinstellung dieser beiden, einander auschlie.enden Linker-Optionen ist EXTDICTIONARY. Wenn Sie keine dieser Optionen angeben, wird EXTDICTIONARY angenommen. Da EXTDICTIONARY die Vorgabe ist, brauchen Sie diese Option nur dann angeben, wenn Sie eine vorherige NOEXTDICTIONARY-Einstellung (z.B. in RTLINKCMD) .berschrieben werden soll. Die Abk.rzungen f.r diese beiden Optionen sind EX bzw. NOE.

Das Extended Dictionary ist ein Liste mit Symbolpositionen und -abh.ngig-keiten innerhalb einer Library-Datei, die von .RTLink zur Beschleunigung der Suche in Library-Dateien verwendet wird. EXTDICTIONARY veranla.t den .RTLink, das Extended Dictionary zu verwenden.

NOEXTDICTIONARY veranla.t den Linker, kein Extended Dictionary zu verwenden. Sie m.ssen diese Option angeben, um die Ausgabe von Warnungen zu vermeiden, die bei zwei m.glcihen Qellen f.r dasselbe PUBLIC-Symbol (z.B. bei der Redefinition einer existierenden Library-Routine in Ihrem Programm).

STACK **Kontrolle der Stack-Gr..e**

STACK gibt eine enue STACK-Gr..e f.r ein Programm an, wobei die alte Gr..e aus dem Objekt-Modul .berschrieben wird. Die Abk.rzung f.r STACK ist ST.

Syntax

/STACK:<ByteAnzahl>

<ByteAnahl> ist die Anzahl der dem Stack zuzuordnenden Bytes und kann jede positive dezimale Zahl bis einschlie.lich 65.535 sein.

Sie k.nnen STACK sowohl im Pre-Link- als auch im Link-Modus angeben. Bei Verwendung im Pre-Link-Modus wird die angegebene Stack-Gr..e alle nachfolgenden, abh.ngigen .EXE-Dateien benutzt, es sein denn, Sie heben diese Einstellung (mit der STACK-Option) bei der Generierung der .EXE-Dateien wieder auf.

EXCLUDE **Ausschlu. von Modulen vom Link-Proze.**

Syntax

/EXCLUDE:<Symbol>
EXCLUDE <SymbolListe>

EXCLUDE bewirkt, da. jedes Modul, in dem das angegebnen Symbol enthalten ist, vom Link-Prozess ausgeschlossen wird. Selbst wenn in einem anderen Modul ein Bezug auf solche Symbole erfolgt (auch durch die Option REFER), wird das Modul nicht mitgelinkt.

Ein Einsatzbeispiel f.r EXCLUDE w.re beispielsweise gegeben, wenn Sie eine Routine oder Funktion, die in einer, der mit Clipper 5.0 ausgelieferten Library-Dateien definiert ist, in Ihrem Programm unter gleichen Namen aber mit ver.ndertem Inhalt verwenden m.chten. Wenn die .EXE-Datei, die dann den Code der neuen Routine enth.lt, zum Beispiel von einer .PLL-Datei abh.ngig ist, liegt ein potentielles Problem vor.

PLL **Angabe einer Pre-Link-Library-Datei (.PLL)**

Syntax

/PLL:<PreLinkLibrary>

PLL wird im Link-Modus dazu verwendet, eine Pre-Link-Library-Datei anzugeben, von der eine zu erzeugende ausf.hrbare .EXE-Datei abh.ngen soll. Wenn Sie diese Option im Pre-Link-Modus verwenden, wird eine .PLL-Datei erzeugt, die wiederum von einer anderen .PLL-Datei abh.ngig ist. Diese Verschachtelung ist aber nur auf eine weitere Ebene beschr.nkt. Sie k.nnen also beispielsweise B.PLL erzeugen, die von B.PLL abh.ngig ist. Es ist aber nicht m.glich, eine weitere Datei C.PLL zu generieren, die von B.PLL abh.ngig ist. Die Abk.rzung von PLL ist PL.

Zur Erstellung der abh.ngigen .EXE-Dateien ben.tigt PLL nur die .PLT-Datei. Die Pre-Link-Library-Datei mu. f.r den Link-Prozess nicht vorhanden sein. Die PLL-Datei ist nur f.r die Ausf.hrung der abh.ngigen .EXE-Datei wichtig.

*** Achtung ***

Die <PreLinkLibrary> sollte immer ohne Erweiterung angegeben werden, da die Option PLL immer die Erweiterung .PLT voraussetzt. Die abh.ngige .EXE-Datei geht dementsprechend ebenfalls von der Erweiterung .PLL aus. Aus diesem Grund sollten Sie niemals die vorgegebenen Dateinamens-Erweiterungen f.r Transfer- und Pre-Link-Library-Dateien .ndern.

PRELINK **Aktivierung des Pre-Link-Modus**

Durch PRELINK wechselt .RTLink vom Link- in den Pre-Link-Modus. Bei Angabe dieser Option wird statt einer .EXE-Datei eine Pre-Link-Library-Datei als <Ausgabedatei> erzeugt. Die vorgegebene Erweiterung f.r eine solche Datei ist .PLL. Au.erdem wird im Pre-Link-Modus eine Transfer mit demselen Namen, aber der Erweiterung .PLT erzeugt. Alle weiteren Eingabeinformationen f.r den Linker entsprechen dem vorgegebenen Link-Modus. Die Abk.rzung f.r PRELINK ist PR.

*** Hinweis ***

Sie k.nnen die Ausgabe von nicht definierten Symbolnamen w.hrend des Pre-Link-Prozesses ignorieren. Wenn allerdings w.hrend der Erstell ung einer .EXE-Datei nicht definierte Symbole angezeigt werden, sollten diese nicht ignoriert werden, im Gegenteil: Ihre Applikation wird bei Aufruf eines nicht definierten Symbols abst.rzen. Das Auftreten von nicht definierten Symbolen w.hrend des Pre-Link-Prozesses ist normal, beim (entg.ligen) Linken m.ssen aber alle Symbole aufgel.st werden bzw. definiert sein. Ist dies nicht der FAll, so bricht .RTLink den Link-Prozess ab und liefert den DOS-R.ckgabecode 1.

REFER **Einbeziehung von Modulen in den Link-Prozess**

Synax

/REFER:<Symbol>
REFER <SymbolListe>

Im Pre-Link-Modus veranla.t REFER den Linker, die angegebenen Symbole mit einzubinden, auch wenn weder eine Bezugnahme auf dieses Modul, noch eine expliziete Definition desselben erfolgt ist. .RTLink durchsucht in diesem Fall alle spezifizierten Library-Dateien, um den notwendigen Code zur Definition dieser Module mit einzubinden. Die Abk.rzung f.r REFER ist REF.

Im Link-Modus wird ein Modul als undefined(nicht definiert) gekennzeichnet, indem es innerhalb einer geladenen Objektdatei als EXTERNAL bezeichnet wird. Im Pre-Link-Modus dagegen werden die einzuschlie.enden Library-Module durch Auflistung mit der Linker-Option REFER einbezogen. Dies erfolgt in der Regel f.r oft benutzte, allgemeine Module, so da. der Code nicht in jeder .EXE-Datei enthalten sein mu., die von einer .PLL-Datei abh.ngig ist.

Sie k.nnen im FREEFORMAT beliebig viele, durch Kommata voneinander getrennte Symbolnamen nach Angabe der Option REFER auflisten. Im Positional-Modus hingegen mu. f.r jedes einzuschlie.ende Symbol erneut die Option /REFER angegeben werden, falls mehrere Symbole bei einem einzigen Link-Prozess mit .RTLink ausgeschlossen werden soll.

BIN2L() Wandelt eine 32-Bit Zahl mit Vorzeichen in einen numerischen Wert um

---

Syntax

BIN2L(<cSignedInt>) -> nWert

Parameter

<cSignedInt> ist eine 32-Bit Zahl mit Vorzeichen in Form einer Zeichenfolge mit 4 Zeichen, wobei das niederwertigste Byte dem ersten Zeichen entspricht. Werden mehr als vier Zeichen angegeben, wird der Rest ignoriert.

R.ckgabe

Numerischer Wert im Integer-Format.

Anwendung

BIN2L() wird in Verbindung mit FREAD() benutzt, um eine 4-Byte Zeichenkette, formatiert als 16-Bit Integer-Zahl mit Vorzeichen, in einen numerischen Wert zu konvertieren. Das ist immer dann n.tzlich, wenn Sie fremde Datentypen lesen und numerische Datentypen verarbeiten wollen, die in ihrer urspr.nglichen Form enthalten sind.

Beispiele

Dieses Beispiel .ffnet eine Datenbank unter Verwendung von systemnahen Dateifunktionen und liest die Anzahl der Datens.tze (Bytes 4-7) ein. Das Ergebnis ist das gleiche wie das von LASTREC().

```
#include "FileIO.ch"
//
DateiNr   := FOPEN("verkauf.dbf", FO_READ)
//
// Setzen des Zeigers auf Byte 4
FSEEK(DateiNr, 4, FS_SET)
//
// Lesen der Anzahl der Datens.tze
SatzZahl  := SPACE(4)
FREAD(DateiNr, @SatzZahl, 4)
//
? BIN2L(SatzZahl) // Resultat: 84
FCLOSE(DateiNr)
```

Dateien

Bibliothek ist EXTEND.LIB, Quelldatei ist EXAMPLEA.ASM

See Also:

BEGINAREA | ENDAREA Bestimmen eines statischen Oberlay-Bereiches

---

Syntax

```
BEGINAREA
   .
   . <Anweisungen>
   .
ENDAREA
```

BEGINAREA wird im Link-Modus verwendet, um den Anfang eines Bereiches mit statischen Overlays zu markieren. Das Bereichsende wird mit ENDAREA festgelegt. Alle zwischen diesen beiden Optionen festgelegten Sektionen (SECTION) werden zu OVerlay-Sektionen innerhalb des Overlay-Bereiches.

Overlay-Bereiche f.r statische Overlays k.nnen durch Angabe einer weiteren BEGINAREA..ENDAREA-Struktur innerhalb einer solchen verschachtelt werden. Die Abk.rzungen f.r BEGINAREA und ENDAREA sind BE und EN.

*** Hinweis ***

Die in diesem Abschnitt behandelte Option ist nur im FREEFORMAT-Modus verf.gbar. Es gibt keine entsprechende Option f.r die Positional-Schnittstelle.

MODULE .bernahme von Modulen in die statische Overlay-Sektion

---

Syntax

MODULE <ModulNamenListe>

MODULE .bernimmt die Segmente der angegebenen Module in die aktuelle statische Overlay-Sektion. MODULE .berschreibt die Optionen FILE und LIBRARY. Die 'Hauptaufgabe von MODULE ist es, (residenten) Code aus Library-Dateien in Overlay-Code umzuwandlen. Die Abk.rzung f.r MODULE ist MO.

Die Namen in der <ModulNamenListe> m.ssen durch Kommata voneinander getrennt werden.

*** Hinweis ***

Die in diesem Abschnitt behandelte Option ist nur im FREEFORMAT-Modus verf.gbar. Es gibt keine entsprechende Option f.r die Positional-Schnittstelle.

PRELOAD **Vorheriges Laden einer Statischen Overlay-Sektion**

PRELOAD bewirkt, da. die aktuelle statische Overlay-Sektion vor Ausf.hrung des Programmes in den Speicher geladen wird. Normalerweise werden nur residente Teile einer Applikation in den Speicher geladen. Die Abk.rzung von PRELOAD ist PRE.

PRELAOD wird nach der vorab zu ladenen SECTION ausgagegeben, z.B.:

```
BEGINAREA
  SECTION = Bereich1
  PRELAOD
  FILE Test1, Test2
  SECTION Bereich2 INTO Extern
  PRELAOD
  FILE Test3
ENDAREA
```

*** Hinweis ***

Die in diesem Abschnitt behandelte Option ist nur im FREEFORMAT-Modus verf.gbar. Es gibt keine entsprechende Option f.r die Positional-Schnittstelle.

SECTION Erzeugen einer statischen Overlay-Sektion

Syntax

SECTION [=<SektionsName>] [INTO <OverlayDate>]

SECTION erzeugt eine neue Sektion f.r statische Overlays und bewirkt, da. Segmente aus nicht mit Clipper compilierten Modulen, die in nachfolgenden FILE- oder LIBRARY-Optionen mit aufgef.hrt sind, Teile dieser Overlay-Sektion werden. Die Abk.rzung f.r SECTION ist SEC.

Wenn ein Clipper-Modul f.r eine Overlay-Sektion angegeben ist, wird genauso verfahren, als l.ge die Option DYNAMIC <OverlayDatei> vor, d.h. Clipper-Module werden als Overlays **immer** dynamisch gelinkt. Soweit Clipper-Module betroffen sind, ist die Struktur einer SECTION oder eines Bereiches f.r statische Overlays bedeutungslos, au.er da. Clipper-Module in der spezifizierten <OverlayDatei> abgelegt werden.

Eine Overlay-Sektion kann in einer seperaten Overlay-Datei abgelegt werden, indem der Zusatz INTO <OverlayDatei> angegeben wird. Die vorgegebene Dateinamenserweiterung ist .OVL. Eine solche Datei wird oft als Externes Overlay bezeichnet. Wenn dieser Zusatz nicht vorhanden ist, wird die Sektion Teil der .EXE-Datei. In diesem Fall spricht amn von einem Internen Overlay.

*** Hinweis ***

Die in diesem Abschnitt behandelte Option ist nur im FREEFORMAT-Modus verf.gbar. Es gibt keine entsprechende Option f.r die Positional-Schnittstelle.

Der Aufruf von RMAKE.EXE

Wenn Sie das Clipper 5.0 Entwicklungssystem entsprechend der Voreinstellungen installiert haben, befindet sich das Programm RMAKE.EXE im Verzeichnis \CLIPPER5\BIN. RMAKE verf.gt generell .ber die folgende Syntax:

RMAKE [<Make-Dateiliste>]
  [<Makro-Definitionsliste>]
  [<Optionen Liste>]

<Make-Dateiliste> gibt eine oder mehrere, durch ein Leerzeichen getrennte Make-Dateien an. Wird als Bestandteil des Dateinamens keine Extension angegeben, so wird .RMK vermutet. Jeder Dateiname kann au.erdem eine Laufwerksangabe und einen Pfad enthalten.

<Makro-Definitionsliste> legt eine oder mehrere Makro-Definitionen in der Form <Makroname>=<Wert> an. Makro-Definitionen m.ssen untereinander durch Leerzeichen getrennt werden. Alle Makros, die .ber diesen Weg definiert wurden, haben Vorrang vor denen, die innerhalb einer Make-Datei angelegt werden.

<Optionen Liste> gibt eine oder mehrere RMAKE-Optionen an. Alle Optionen k.nnen sowohl mit kleinen als auch mit gro.en Buchstaben angegeben werden und m.ssen jeweils durch einen Slash (/) oder durch einen Bindestrich (-) eingeleitet werden. Dar.ber hinaus wird zwischen den Optionen kein anderes Separationszeichen ben.tigt.
Bei Aufruf ohne irgendwelche Argumente zeigt RMAKE eine Liste aller verf.gbaren Optionen mit kurzer Beschreibung am Bildschirm an.

Die MAKE-Datei

Fundamentaler Bestandteil eines Make-Systems ist die Make-Datei, eine ASCII-Textdatei, die mit Hilfe eines normalen Text-Editors angelegt werden kann. Eine solche Make-Datei legt fest, welche Dateien aktualisiert werden sollen und was zu einer solchen Aktualisierung getan werden mu., wenn diese nicht .ber das korrekte Datum und die korrekte Uhrzeit verf.gen. In einem typischen Clipper-Programm w.rden demnach alle Quell- und Header-Dateien angegeben, aus denen die Objekt-Dateien resultieren, sowie in einer weiteren Auflistung alle Objekt-Dateien, aus denen letztendlich die ausf.hrbare Datei resultiert. In Verbindung mit der Auflistung dieser Dateien werden jeweils alle Anweisungen angegeben, die zum Erstellen aktueller Versionen der Zieldateien notwendig sind. Eine Make-Datei enth.lt die folgenden, grundlegenden Elemente:

. Abh.ngigkeitsregeln, die festlegen, wie eine Datei zu erzeugen ist

. Schlu.folgerungsregeln, die festlegen, was zu tun ist, falls bei einer Abh.ngigkeitsregel keine Anweisungen angegeben wurden.

. Makros, die es erlauben, in den verschiedenen Regeln auch variable Werte zu verwenden.

. Direktiven, die eine zus.tzliche Kontrolle des Ablaufs einer Make-Datei erlauben.

Jede dieser Komponenten wird im folgenden beschrieben.

Die Benutzung von Anf.hrungszeichen

In bestimmten Situationen sind unter RMAKE Anf.hrungszeichen notwendig, damit verschiedene Symbole nicht als Anweisungen interpretiert werden. Dabei sind immer sowohl einfache als auch doppelte Anf.hrungszeichen erlaubt.

Wenn Sie beispielsweise innerhalb einer Abh.ngigkeits- oder Schlu.folgerungsregel einen Dateinamen in Verbindung mit einem Pfad angeben, so k.nnten sowohl der Doppelpunkt als auch der Backslash (\) zu Problemen f.hren, da beide Zeichen unter RMAKE eine besondere Bedeutung haben. Aus diesen Gr.nden mu. eine Pfad-Spezifikation in Anf.hrungszeichen angegeben werden. Ein anderer Fall sind Definitionen von Makronamen, welche die gleichen oder andere spezielle Symbole enthalten. Da derart angegebene Anf.hrungszeichen nie selber interpretiert werden, sollten sie grunds.tzlich immer dann eingesetzt werden, wenn Zweifel bestehen.

Hinweis
Die Verwendung von Anf.hrungszeichen beeinflu.t nicht die Art und Weise, in der RMAKE das Makrozeichen $ behandelt. Makros werden innerhalb von Anf.hrungszeichen genauso wie au.erhalb dieser expandiert.

Die Fortsetzung einer Zeile

Jeder Kommentar, jede Regel und jede Anweisung innerhalb einer Make-Datei kann in der n.chsten Zeile fortgesetzt werden, indem ein Backslash (\) an das Ende der fortzusetzenden Zeile gestellt wird. Befindet sich ein Kommentar auf einer derart fortgesetzten Zeile, so mu. dieser dem Fortsetzungssymbol (\) folgen. Ein Beispiel:

```
TEST.OBJ: Test1.prg, Test2.prg, Test3.prg\
Test4.prg
  CLIPPER @Test.clp
```

Kommentare

Kommentare k.nnen innerhalb einer Make-Datei, in gleicher Weise wie bei C, als Multiline- (/* ... */) oder als Inline- (//) Kommentare angegeben werden. Damit kann ein Kommentar in einer Make-Datei f.r sich selber stehen oder am Ende einer Befehlszeile innerhalb einer Abh.ngigkeitsregel angegeben werden. Ein Beispiel:

```
/* Dies ist eine g.ltige Kommentar-Zeile */
TEST.OBJ: Test.prg
  CLIPPER Test.prg  // Ein g.ltiger Inline-Kommentar
```

Wurde die Option /U angegeben, so kann das hash-Zeichen (#) zur Einleitung eines Kommentares verwendet werden. Kommentare k.nnen grunds.tzlich nicht verschachtelt werden.

Abh.ngigkeitsregeln

Entscheidender Bestandteil einer Make-Datei ist die Abh.ngigkeitsregel, die aus einem Abh.ngigkeits-Statement, gefolgt von einer oder mehreren Anweisungen besteht. Das Abh.ngigkeits-Statement formuliert dabei die Beziehung zwischen einer Zieldatei und einer oder mehreren abh.ngigen Dateien. Wenn eine der abh.ngigen Dateien .ber ein Datum oder eine Uhrzeit j.nger als die Daten der Zieldatei verf.gt, so werden alle der Regel folgenden Anweisungen ausgef.hrt, um so die Zieldatei auf den neuesten Stand zu bringen.

Eine Abh.ngigkeitsregel verf.gt grunds.tzlich .ber den folgenden Aufbau:

```
<Zieldatei>: <Liste abh.ngiger Dateien>
[<Anweisung>]
[<Anweisung>]
...
```

Jeder Dateiname, der Teil einer Abh.ngigkeitsregel ist, kann zusammen mit dem jeweiligen Laufwerk und Pfad angegeben werden. Wurden weder Laufwerk noch Pfad angegeben, so sucht RMAKE die Dateien im angemeldeten oder in den .ber das makePath-Makro angegebenen Pfaden (s. Abschnitt Wie RMAKE Dateien sucht in diesem Kapitel). Anf.hrungszeichen k.nnen benutzt werden, falls ein Dateiname oder Pfad bestimmte, unter RMAKE reservierte Zeichen enth.lt. Da zum Beispiel der Doppelpunkt innerhalb der Abh.ngigkeitsregel .ber eine besondere Bedeutung verf.gt, sollten Dateinamen mit Laufwerksangabe immer in Anf.hrungszeichen gesetzt werden. Die Extensionen m.ssen ebenfalls Bestandteil der Dateinamen sein.

<Zieldatei> ist der Name der Datei, die gegebenenfalls neu erzeugt werden soll, gefolgt von einem Doppelpunkt und einem Leerschritt. Im folgenden Beispiel wird die Zieldatei im angemeldeten Verzeichnis angelegt:

```
MEINPROG.EXE: MEINPROG.OBJ "\CLIPPER5\OBJ\IBMBIOS.OBJ"
```

w.hrend die folgende Zieldatei im angegebenen Verzeichnis angelegt wird:

```
"C:\PROGS\MEINPROG.EXE": MEINPROG.OBJ "\CLIPPER5\OBJ\IBMBIOS.OBJ
```

<Liste abh.ngiger Dateien> ist eine Liste, bestehend aus einer oder mehreren Dateien, die durch Leerschritte getrennt werden m.ssen. Im folgenden Beispiel sind zwei abh.ngige Dateien angegeben:

```
MEINPROG.EXE: MEINPROG.OBJ "\CLIPPER5\OBJ\IBMBIOS.OBJ"
```

RMAKE sucht im angemeldeten Verzeichnis nach MEINPROG.OBJ und in dem explizit angegebenen Verzeichnis \CLIPPER5\OBJ auf dem angemeldeten Laufwerk nach der Datei IBMBIOS.OBJ.

F.r die <Anweisung>-Statements k.nnen eine oder auch mehrere g.ltige DOS-Kommandozeilen angegeben werden. Jede Anweisung mu. in einer separaten Zeile stehen und entweder durch Leerzeichen oder Tabulatoren gegen.ber dem linken Rand einger.ckt sein. RMAKE geht davon aus, da. ein Anweisungsblock in der ersten leeren oder nicht mehr einger.ckten Zeile endet.

Alle Anweisungen des jeweiligen Anweisungsblocks werden ausgef.hrt, wenn eine der Dateien aus <Liste abh.ngiger Dateien> .ber ein neueres Datum oder eine neuere Uhrzeit als die <Zieldatei> verf.gt. Wurden in Verbindung mit einer Abh.ngigkeitsregel keinerlei Anweisungen angegeben, so sollte sich an beliebiger Stelle dieser Make-Datei eine Schlu.folgerungsregel befinden, welche die zu verwendenden Anweisungen definiert. Wenn zum Beispiel eine Zieldatei TEST.OBJ von einer Datei TEST.PRG abh.ngig ist, so sollte die Make-Datei .ber Anweisungen zur erneuten Compilierung verf.gen, falls die Quell-Datei neuer als die Obj-Datei

ist. Die folgende Abh.ngigkeitsregel erledigt genau dies:

```
TEST.OBJ: Test.prg
  CLIPPER Test
```

Mit einer .hnlichen Regel kann auch die .EXE-Datei erstellt werden. Um eine Liste abh.ngiger Dateien zu demonstrieren, wird im folgenden Beispiel davon ausgegangen, da. WINDOW.OBJ f.r das Erstellen der ausf.hrbaren Datei ebenfalls mit eingebunden werden mu.. Die Regel besagt, da. eine neue .EXE-Datei zu erstellen ist, falls entweder TEST.OBJ oder WINDOW.OBJ .ber ein neueres Datum als diese verf.gen:

```
TEST.EXE: Test.obj Window.obj
  RTLINK FI Test, Window
```

Schlu.folgerungsregeln

Eine Schlu.folgerungsregel kann eine Reihe von Anweisungen f.r solche Abh.ngigkeits-Statements festlegen, die selber .ber keinen Anweisungsblock verf.gen. Dies kann als eine Art Abk.rzung eingesetzt werden, wenn die selben Anweisungs-Bl.cke mehrfach f.r verschiedene Abh.ngigkeits-Statements ben.tigt werden. Der grunds.tzliche Aufbau einer Schlu.folgerungsregel lautet folgenderma.en:

```
.<Abh.ngigeExtension>.<ZielExtension>
```

Im Gegensatz zu einem Abh.ngigkeits-Statement sind bei einer Schlu.folgerungsregel die Reihenfolge der abh.ngigen und der Zieldateien umgekehrt.

Wenn RMAKE auf ein Abh.ngigkeits-Statement st..t, dem keine Anweisungen folgen, sucht es nach nach einer Schlu.folgerungsregel, welche die Extensionen der abh.ngigen und der Zieldatei enth.lt. Wird eine passende Schlu.folgerungsregel gefunden, so f.hrt RMAKE die zugeh.rigen Anweisungen einschlie.lich der Expansion entsprechender Makros aus. Wie auch in einer Schlu.folgerungsregel mu. jede Anweisung in einer separaten Zeile angegeben und entweder durch Leerzeichen oder durch Tabulatoren einger.ckt sein. Die Liste der Anweisungen endet wiederum bei der n.chsten leeren oder nicht einger.ckten Zeile.

Bei der Suche nach einer Schlu.folgerungsregel sind die Dateinamen aus einem Abh.ngigkeits-Statement ohne Belang - es werden nur die Extensionen verglichen. Ist in einem Abh.ngigkeits-Statement mehr als eine abh.ngige Datei angegeben, so wird die Extension der ersten Datei dieser Liste benutzt. Wenn die Extensionen .bereinstimmen, wird die Regel durchgef.hrt.

Die Schlu.folgerungsregel im folgenden Beispiel sorgt daf.r, da. eine .PRG-Datei aus einem Abh.ngigkeits-Statement zu einer .OBJ-Datei compiliert wird:

```
.prg.obj:
  CLIPPER $** /M
```

Das $** Symbol in der Kommandozeile des Compilers ist ein vordefiniertes Makro von RMAKE, das im n.chsten Abschnitt beschrieben wird. Grunds.tzlich sorgt es daf.r, da. der Name der jeweiligen abh.ngigen Datei an dieser Stelle eingesetzt wird, in unserem Beispiel also zum Kommandozeilen-Argument des Compilers wird. Aus dem folgenden Statement:

```
TEST.OBJ: Test.prg
```

w.rde demnach die Anweisung CLIPPER Test.prg /M resultieren. Wird innerhalb der Anweisungs-Liste einer Abh.ngigkeits- oder Schlu.folgerungsregel ein DOS SET-Kommando gefunden, so wird dies im Gegensatz zu anderen Anweisungen nicht zur Ausf.hrung an den DOS Kommando-Interpreter .bergeben, sondern direkt durch RMAKE interpretiert. Dies erlaubt das Setzen von Umgebungs-Variablen als Teil eines Anweisungsblockes. Dazu ein Beispiel:

```
.prg.obj:
  SET CLIPPERCMD=$(Attributliste)
  CLIPPER $**
```

Direktiven

F.r die Anwendung innerhalb von Make-Dateien werden mehrere Direktiven zur Verf.gung gestellt, die w.hrend der Parser-Phase ber.cksichtigt werden. Jede Direktive mu. in der Make-Datei auf einer separaten Zeile angegeben sein, und sie k.nnen nicht als Teil einer Abh.ngigkeits- oder Schlu.folgerungsregel benutzt werden. Alle Makros, die als Argumente f.r Direktiven angegeben wurden, werden sofort expandiert.

Die Syntax aller Direktiven in diesem Bereich umfa.t ein einleitendes hash-Zeichen (#), das jeweils angegeben werden mu.. Beachten Sie bitte, da. sowohl das Ausrufungszeichen (!) als auch das erw.hnte hash-Zeichen (#) abwechselnd benutzt werden d.rfen, es sei denn, die Option /U wurde ausgew.hlt. In diesem Fall mu. das Ausrufungszeichen (!) verwendet werden, da alle Zeilen, die mit einem hash-Zeichen (#) beginnen, sonst wie ein Kommentar behandelt w.rden.

Bedingte Direktiven

Bedingte Direktiven gestatten es Ihnen, abh.ngig von der Existenz oder dem Wert eines Makros, Teile einer Make-Datei von der Bearbeitung auszuschlie.en. Ein solches Makro kann entweder zuvor in der Make-Datei angelegt, in der RMAKE-Kommandozeile angegeben oder in der Umgebungs-Variablen RMAKE .ber die Option /D festgelegt werden. Die Struktur einer bedingten Direktive besteht aus der #if-Direktive, gefolgt von einer variablen Anzahl von Statements. Abgeschlossen wird ein solches Konstrukt durch die Direktive #endif. Durch Verwendung der Direktive #else kann ein alternativer Block mit Statements festgelegt werden, der dann zur Ausf.hrung kommt, wenn die Bedingung f.r das #if nicht zutrifft.

#ifdef Ausf.hren, wenn das Makro existiert

```
#ifdef <Makroname>
  <Statements>...
[#else
  <Statements>]...
#end[if]
```

Ein #ifdef l..t die Durchf.hrung aller folgenden Statements zu, wenn das angegebene Makro existiert. Das Dollarzeichen ($) darf in einem solchen Falle nur dann am Beginn des Makronamens angegeben werden, wenn die Existenz des Makros .berpr.ft werden soll, dessen Name wiederum aus der Expansion des spezifizierten Makros resultiert. Ein Beispiel:

```
#ifdef pfadNeu
  makepath[.prg] := "$(pfadNeu);$(makepath[.prg])"
#endif
```

#ifndef Ausf.hren, wenn das Makro nicht existiert

```
#ifndef <Makroname>
  <Statements>...
[#else
  <Statements>]...
#end[if]
```

Die Direktive #ifndef gleicht in ihrer Wirkungsweise dem #ifdef, nur da. die unmittelbar folgenden Statements nur dann ausgef.hrt werden, wenn das angegebene Makro nicht existiert. Dazu ein Beispiel:

```
#ifndef "makepath[.prg]"
  makepath[.prg] := "$(src)"
#endif
```

#ifeq Ausf.hren, wenn zwei Begriffe identisch sind

```
#ifeq <Wort1> <Wort2>
```

```
    <Statements>...
  [#else
    <Statements>...]
  #end[if]
```

Die Direktive #ifeq f.hrt alle unmittelbar folgenden Anweisungen aus, wenn die zwei angegebenen Begriffe identisch sind. Diese Begriffe m.ssen in Anf.hrungszeichen (") angegeben werden, falls sie irgendwelche Leerzeichen enthalten, da sie durch Leerzeichen voneinander getrennt werden. Beachten Sie bitte, da. der Vergleich unabh.ngig von der Gro.-/Kleinschreibung (nicht Case-sensitive) ist. Ein Beispiel dazu:

```
  #ifeq $(os) windows
  .obj.exe:
    RTLINK $**, $@;
  #endif
```

#iffile Ausf.hren, wenn eine Datei existiert

```
  #iffile <Dateiname>
    <Statements>...
  [#else
    <Statements>...]
  #end[if]
```

#iffile f.hrt alle folgenden Anweisungen aus, wenn die angegebene Datei existiert. Wildcards (Joker-Zeichen, *?) sind in <Dateiname> erlaubt, so da. Sie zum Beispiel auch die Existenz einer Gruppe von Dateien in einem Verzeichnis pr.fen k.nnen. Nur den Namen eines Verzeichnisses anzugeben, wird hingegen nicht funktionieren. Ein Beispiel:

```
  #iffile "C:\MEINPROG\*.*"
    prepath="C:\MEINPROG"
  #endif
```

Ber.cksichtigen Sie bitte, da. solche Anweisungen w.hrend der Parser-Phase durchgef.hrt werden und demnach nicht Teil einer Anweisung sein k.nnen. Au.erdem werden makepath-Makros bei der Suche nach Dateien durch ein #iffile nicht ber.cksichtigt.

#undef Die Definition eines Makros aufheben

```
  #undef <Makroname>
```

.ber diese Direktive wird jede vorherige Definition eines Makros, einschlie.lich der in einer Kommandozeile von RMAKE durchgef.hrten, wieder aufgehoben. Es ist jedoch nicht m.glich, ein Makro aus der Umgebungs-Variablen RMAKE wieder aufzuheben. Ein solches Makro kann jedoch versteckt werden, indem eines mit gleichem Namen angelegt wird, und wieder sichtbar gemacht werden, indem das angelegte Makro .ber #undef gel.scht wird.

#include Include-Dateien

```
  #include "<Dateiname>"
```

Anstelle der #include-Direktive wird der Inhalt der angegebenen Datei bei Abarbeitung der Make-Datei geladen und ausgef.hrt. Der Dateiname darf sowohl einen Pfad als auch eine Datei-Extension enthalten, wobei keinerlei Voreinstellungen ber.cksichtigt werden. Ein Beispiel:

```
  #include "RMAKE.INI"
```

Die #include-Direktive beachtet weder die Umgebungs-Variable INCLUDE (Clipper-Compiler), noch einen Pfad aus dem makepath-Makro, wenn sie nach Dateien sucht. Wird kein Pfad explizit angegeben, so wird ausschlie.lich im angemeldeten Verzeichnis gesucht.

#stderr Die Anzeige einer Fehlermeldung

```
#stderr "<text>"
```

Schreibt die Meldung <text> in die aktuelle Fehler-Datei oder -Einheit (Error-Device). Zum Beispiel:

```
#stderr "Fehler beim Compilieren!"
```

#stdout Anzeigen eines Textes

```
#stdout "<text>"
```

Gibt den <text> an die Standard Ausgabe-Datei oder -Einheit (Output-Device) aus. Als Beispiel:

```
#stdout "Generieren einer neuen Version ..."
```

#! Ausf.hren einer Anweisung

```
#! <Anweisung>
```

F.hrt die angegebene Anweisung w.hrend der Parser-Phase unmittelbar aus. Wie bei allen anderen Anweisungen, die in Verbindung mit Regeln angegeben wurden, kann hier jedes g.ltige DOS-Kommando mit seinen Argumenten angegeben werden. Dazu ein Beispiel:

```
#!DIR *.OBJ
```

BIN2W() Wandelt eine 16-Bit Zahl ohne Vorzeichen in einen numerischen Wert um

---

Syntax

BIN2W(<cUnsignedInt>) -> nWert

Parameter

<cUnsignedInt> ist die vorzeichenlose 16-Bit Zahl in Form einer Zeichenfolge von 2 Zeichen, wobei das niederwertigste Byte dem ersten Zeichen entspricht. Werden mehr als zwei Zeichen angegeben, wird der Rest ignoriert.

R.ckgabe

Numerischer Wert im Integer-Format.

Anwendung

BIN2W() wird in Verbindung mit FREAD() benutzt, um eine 2-Byte Zeichenkette, formatiert als 16-Bit Integer-Zahl ohne Vorzeichen, in einen numerischen Wert zu konvertieren. Das ist immer dann n.tzlich, wenn Sie fremde Datentypen lesen und numerische Datentypen verarbeiten wollen, die in ihrer urspr.nglichen Form enthalten sind.

Beispiele

Dieses Beispiel .ffnet eine Datenbank unter Verwendung von systemnahen Dateifunktionen und liest die Gr..e der Datens.tze (Bytes 10-11) ein. Das Ergebnis ist dasselbe wie das von RECSIZE().

```
#include "FilerIO.ch"
//
DateiNr   := FOPEN("Verkauf.dbf", FO_READ)
//
// Setzen des Zeigers auf Byte 10
FSEEK(DateiNr, 10, FS_SET )
//
// Lesen der Gr..e der Datens.tze
RecLen    := SPACE(2)
FREAD(DateiNr, @RecLen, 2)
//
? BIN2L(RecLen) // Resultat: 124
FCLOSE(DateiNr)
```

Dateien

Bibliothek ist EXTEND.LIB, Quelldatei ist EXAMPLEA.ASM

See Also:

Makros

Um Ihnen die Benutzung variabler Informationen bei der Festlegung von Abh.ngigkeits- oder Schlu.folgerungsregel zu erlauben, stellt RMAKE Makros zur Verf.gung. Dies erlaubt es Ihnen, eine Folge von Zeichen mit einem Makronamen in Verbindung zu bringen. Wann immer der Makroname dann innerhalb der Make-Datei auftritt, wird er durch die zugeordnete Zeichenfolge ersetzt. Wie Sie vielleicht bemerkt haben, ist dies sehr .hnlich der Pre-Prozessor-Direktive #define, die bei der Compilierung von .PRG-Dateien verwendet wird.

Makros werden in einer Reihe von Situationen einschlie.lich der folgenden eingesetzt:

. Zur Kontrolle der Verzeichnisnamen und Kommandozeilen-Optionen des Compilers und des Linkers f.r unterschiedliche Projekte.

. Um Listen von abh.ngigen Dateien zu erstellen, die mehr als einmal in einer Make-Datei vorkommen.

Unter RMAKE existieren zwei Arten von Makros: die anwenderdefinierten und die vordefinierten, die beide im folgenden beschrieben werden.

Anwenderdefinierte Makros

In einer Make-Datei wird eine Makro-Definition auf einer separaten Zeile in der folgenden Form angegeben:

```
<Makroname>=<Wert>
```

Der <Wert> f.r das Makro kann eine beliebige Folge von Zeichen, angegeben in Anf.hrungszeichen, aber auch eine Null-Zeichenfolge ("") sein. Makro-Definitionen k.nnen auch verschachtelt werden, so da. der Makroname wiederum andere Makros enthalten kann. Dies erlaubt es unter anderem, ein Makro einem anderen zuzuweisen.

Achtung

Ann.hernd jedes Zeichen, mit Ausnahme der folgenden, darf in einem Makronamen verwendet werden: das Dollar-Zeichen ($), Klammern, der Doppelpunkt (:), der Punkt (.) und das Prozent-Zeichen (%).

Um ein Makro nach seiner Definition zu benutzen, verwenden Sie die folgende Form:

```
$(<Makroname>)
```

Sobald RMAKE eine Anweisung der Form $(<Makroname>) findet, ersetzt es diese durch die Folge der zuvor definierten Zeichen. Wurde dieses Makro nicht definiert, so wird eine Null-Zeichenfolge erzeugt. Ein Makro beh.lt so lange seine G.ltigkeit, bis es umdefiniert, seine Definition aufgehoben oder aber RMAKE beendet wird.

Im folgenden Beispiel wird ein Makro mit einer Liste abh.ngiger Dateien angelegt:

```
Dateien=TEST.OBJ TEST1.OBJ TEST2.OBJ
```

Im weiteren Verlauf der Make-Datei kann dieses Makro dann als Ersatz f.r eine Liste von Dateien innerhalb einer Regel angegeben werden:

```
TEST.EXE: $(Dateien)
```

Es existieren zwei Situationen, in denen ein Makro nicht sofort expandiert wird, nachdem es gefunden wurde. Zum einen sind dies Makros in den Anweisungen einer Schlu.folgerungsregel, die erst expandiert werden, wenn eine Verbindung zu einer Abh.ngigkeitsregel hergestellt werden konnte. Zum zweiten sind dies Makros innerhalb von

Makro-Definitionen, die erst expandiert werden, wenn das Makro, in dem sie enthalten sind, expandiert wird. Ein Beispiel:

```
Makro1 = $(cDatei)  // cDatei ist noch nicht definiert
cDatei = XDatei.prg // cDatei wird definiert
XDATEI.OBJ: $Makro1 // XDATEI.OBJ abh.ngig von XDATEI.PRG
```

Eine derart verz.gerte Expansion erm.glicht es einem Kommandozeilen-Makro, sich auf ein Makro in einer Make-Datei zu beziehen, sofern das Make-Makro vor der Benutzung des Kommandozeilen-Makros definiert wird. Da dieses Verhalten auch zu Komplikationen f.hren kann, ist es m.glich, Makros so zu definieren, da. diese immer sofort expandiert werden. Dazu mu. nur der immedaite Zuweisungs-Operator (:=) anstelle des einfachen Gleichheits-zeichens eingesetzt werden. Ein Beispiel:

```
Makro1 := $(cDatei) // Funktioniert nur, wenn cDatei existiert
```

Hinweis
Makros k.nnen in der RMAKE-Kommandozeile mit der gleichen Syntax wie in einer Make-Datei und in der RMAKE-Umgebungsvariablen .ber die Option /D angegeben werden. Da solche Makros nicht umdefiniert werden k.nnen, haben Makros aus der Kommandozeile oder aus der Umgebungsvariablen Vorrang vor denen, die in einer Make-Datei angelegt wurden.

Vordefinierte Makros
Neben den anwenderdefinierten Makros existiert eine Reihe vordefinierter Makros, .ber die ein Zugriff auf die abh.ngigen und auf die Zieldateien aus der letzten Abh.ngigkeitsregel m.glich ist. Solche vordefinierten Makros werden in der Regel eingesetzt, wenn Anweisungen in Verbindung mit einer Abh.ngigkeits- oder Schlu.folgerungsregel angegeben werden. Eine .bersicht der vordefinierten Makros liefert die folgende Tabelle:

Die folgende Schlu.folgerungsregel benutzt zum Beispiel das $** Makro, um eine einzelne Programm-Datei (.prg), abh.ngig von einer Objekt-Datei (.obj) innerhalb einer zugeh.rigen Abh.ngigkeitsregel, zu compilieren.

```
.prg.obj:
   CLIPPER $** /M
```

Diese Formulierung wird nur in Verbindung mit einer Abh.ngigkeitsregel mit einer einzelnen abh.ngigen Datei funktionieren, da Clipper immer nur eine einzelne Datei compilieren kann, das $**Makro jedoch immer die ganze Liste der abh.ngigen Dateien liefert. Beachten Sie bitte die umfangreichen Hinweise am Ende dieses Kapitels, in denen auch beschrieben wird, wie mehrere Programm-Dateien am besten zu compilieren sind. Au.erdem existiert das zumindest namentlich vordefinierte makepath-Makro, .ber das die Pfade angegeben werden k.nnen, in denen RMAKE Dateien sucht und ablegt.